# The GNUstep cookbook

Marko Riedel

July 21, 2008

Welcome to the GNUstep cookbook. This document provides a variety of different recipes for the GNUstep programmer. Recipes are explained and the source code is included. Recipes are code snippets that the programmer can quickly get a grasp on, without having to work through the complex logic of a complete application. The TGZ archive of this cookbook contains the LaTeX source as well as the source for each recipe, which may be more material than what we discuss in the text. The URL is `http://www.gnustep.it/marko/`. We hope that this cookbook can be of use to you in your everday GNUstep programming needs and encourage submissions that fit the recipe format. Enjoy! (With corrections by Wim Oudshoorn.)

## Contents

**1 Combinatorics** **4**
1.1 Counting I: Subset enumerator . . . 4
1.1.1 Idea . . . 4
1.1.2 Implementation . . . 4
1.1.3 Remark . . . 6
1.2 Counting II: Permutation enumerator . . . 7
1.2.1 Idea . . . 7
1.2.2 Implementation . . . 8
1.2.3 Remark . . . 10
1.3 Steganography . . . 11
1.3.1 Idea . . . 11
1.3.2 Implementation . . . 11
1.4 Working with GMP integers . . . 18
1.4.1 Idea . . . 18
1.4.2 Implementation . . . 19
1.5 Enumerating graphs with Pólya's theorem and GMP . . . 23
1.5.1 Introduction . . . 23
1.5.2 Permutations . . . 24
1.5.3 The Cycle Index . . . 24
1.5.4 Pólya's theorem . . . 25
1.5.5 Cycle index of the pair group . . . 25
1.5.6 Concerning memory allocation . . . 27
1.5.7 Implementation I: auxiliary functions . . . 27
1.5.8 Implementation II: Polynomials . . . 33
1.5.9 Implementation III: Pólya's theorem . . . 38
1.5.10 Results . . . 42
1.6 Backtracking: The n-queens problem . . . 43
1.6.1 Idea . . . 43
1.6.2 Implementation . . . 44

**2 Processing mouse events** **52**
2.1 Selecting a part of a static image . . . 52
2.1.1 Idea . . . 52
2.1.2 Implementation . . . 52

**3 Working with tasks** **54**
3.1 User may interrupt read from task . . . 54
3.1.1 Idea . . . 54
3.1.2 Implementation . . . 54
3.2 Frontend to `df` . . . 57
3.2.1 Idea . . . 57
3.2.2 Implementation . . . 57
3.3 Code browser with `find` and `grep` . . . 66
3.3.1 Idea . . . 66
3.3.2 Implementation . . . 67

**4 Distributed objects** **90**
4.1 Connect four . . . 90
4.1.1 Idea . . . 90
4.1.2 Implementation . . . 90

**5 CGI programming** **101**
5.1 Web server . . . 101
5.1.1 Idea . . . 101
5.1.2 Preliminaries . . . 101
5.1.3 Implementation . . . 102
5.2 Read GET and POST variables from forms and URLs . . . 108
5.2.1 Idea . . . 108
5.2.2 Implementation . . . 108
5.3 Working with cookies . . . 111
5.3.1 Idea . . . 111
5.3.2 Preliminaries . . . 112
5.3.3 Implementation . . . 112
5.4 Webchat with MySQL . . . 117
5.4.1 Idea . . . 117
5.4.2 MySQL tables used . . . 118
5.4.3 Preliminaries . . . 119
5.4.4 Implementation I: auxiliary functions and definitions . . . 119
5.4.5 Implementation II: `main` . . . 126

**6 Puzzles** **139**
6.1 The Sixteen Puzzle . . . 139
6.1.1 Idea . . . 139
6.1.2 Implementation . . . 139

**7 Miscellaneous** **147**
7.1 Color picker with X11 . . . 147
7.1.1 Idea . . . 147
7.1.2 Implementation . . . 147
7.2 Screen grab with X11 . . . 150
7.2.1 Idea . . . 150
7.2.2 `XWinData` . . . 151
7.2.3 `ImageView` . . . 159
7.2.4 `Controller` and `main` . . . 160
7.3 Calendar view . . . 167
7.3.1 Idea . . . 167
7.3.2 Calendar view: interface . . . 168
7.3.3 Calendar view: implementation . . . 170

# LICENCE ALC TYPE 3

## Article 1

La présente licence s'adresse à tout utilisateur de ce document. Un utilisateur est toute personne qui lit, télécharge ou reproduit ce document

## Article 2

La licence rappelle les droits de l'auteur sur son document et les prérogatives qu'il a entendu concéder aux utilisateurs. Chaque utilisateur est donc tenu de lire cette licence et d'en respecter les termes.

## Article 3

La licence accompagne le document de manière intrinsèque. Tout utilisateur qui reproduit le document et le cède à un tiers est tenu de céder la licence. La licence s'impose alors à ce tiers, et ce même en cas de cessions successives. En outre, cette licence ne peut en aucun cas être modifiée.

## Article 4

Toute concession accordée par l'auteur sur ses droits, conformément à l'objet de l'association qui est de promouvoir l'enseignement et le partage des connaissances, ne vaut que tant que l'exploitation du document est faite à titre gratuit. L'utilisation du document dans le cadre d'un enseignement payant ne constitue pas une utilisation à titre commerciale au sens de la présente licence. Le recouvrement de sommes correspondant à des frais de port ou de mise sur support n'a pas pour effet d'attribuer un caractère commercial à l'exploitation du document. Ces frais ne peuvent en aucun cas exéder les sommes déboursées pour ces services.

## Article 5

L'auteur autorise tout utilisateur à reproduire et représenter son oeuvre, sur tout support, dès lors que l'utilisation, privée ou publique, est non commerciale, et sous réserve du respect des dispositions de l'article 6. La reproduction partielle du document est autorisée sous réserve du respect des dispositions de ce même article.

## Article 6

L'utilisateur doit respecter les droits moraux de l'auteur: aussi bien la paternité de l'auteur que l'intégrité de l'oeuvre. Ainsi, le nom de l'auteur doit toujours être clairement indiqué, même en cas de reproduction ou représentation partielle. En cas de reproduction ou représentation partielle, il devra être précisé qu'il ne s'agit que d'un extrait, et référence devra être faite à l'oeuvre intégrale. Aucune modification du document n'est autorisée, à l'exeption des modifications apportées par l'auteur lui-même.

## Article 7

L'utilisateur peut apporter sa contribution (actualisation, remarques, précision...). Ces insertions doivent apparaitre clairement comme telles (emploi de caractères italiques, ou d'une couleur différente) et doivent être datées. Cet utilisateur devient un auteur contributaire et doit à ce titre être identifiable. Sa contribution sera soumise à cette même licence. La Charte des auteurs lui sera alors applicable concernant cette contribution. Le document peut être traduit dès lors qu'il est fait référence au texte original, que l'auteur reste identifié, et qu'il est précisé qu'il s'agit d'une traduction.

## Article 8

Toute infraction à la présente licence pourrait constituer une atteinte aux droits de l'auteur sur son oeuvre. Dans tous les cas, le non-respect de la licence sera suceptible de fonder une action en justice.

## Article 9

En cas de litige, la loi française sera la seule applicable.

# 1 Combinatorics

## 1.1 Counting I: Subset enumerator

*by Marko Riedel*

### 1.1.1 Idea

The goal is to enumerate all subsets of some set of elements stored in an array. It is true for any such subset that an original element from the array is either present or absent. This means that we can enumerate subsets by counting in binary from 0 to $2^n - 1$, where $n$ is the number of elements in the array. If bit $k$ is set, then we include the element at $k$, otherwise it is left out. We could use machine integers to count to $2^n - 1$, but then the size of a machine integer would limit the size of the array whose subsets we wish to enumerate.

### 1.1.2 Implementation

We will implement an enumerator for subsets and add a category to `NSArray`.

```
@interface SubsetEnumerator : NSEnumerator
{
   NSArray *array;
   char *binary;
   int count;
   BOOL done;
}

- (id)initWithArray:(NSArray *)anArray;
- (id)nextObject;
- (void)dealloc;

@end
```

The state of the enumerator is defined by the array whose subsets we wish to enumerate, the number of elements in the array, an array that stores the binary digits of the counter and a flag that indicates whether we are done.

```
@implementation SubsetEnumerator

- (id)initWithArray:(NSArray *)anArray
{
   int index;

   [super init];

   count = [anArray count];
   array = anArray; [array retain];

   binary = NSZoneMalloc(NSDefaultMallocZone(),
                  count*sizeof(char));
   for(index=0; index<count; index++){
      binary[index] = 0;
   }

   done = NO;

   return self;
}
```

The enumerator retains the array. We store the number of elements in the instance variable `count` for easy reference. We allocate space for the binary digits of the counter and set them all to zero.

```
- (id)nextObject
{
   NSMutableArray *result;

   int index;
   if(done==YES){
      return nil;
   }

   result = [NSMutableArray arrayWithCapacity:1];
   for(index=0; index<count; index++){
      if(binary[index]){
         [result
            addObject:
               [array objectAtIndex:index]];
      }
   }
```

The first part of `nextObject` computes the current subset. It checks every binary digit and includes the corresponding item if the bit is set.

```
   for(index=count-1; index>=0; index--){
      if(!binary[index]){
         binary[index]++;
         while(++index<count){
            binary[index] = 0;
         }
         break;
      }
   }
   if(index<0){
      done = YES;
   }

   return result;
}
```

The second part increments the counter, i.e. it implements binary addition. Scan from the right until you find a zero digit, increment this digit and set remaining positions to zero. We are done enumerating when there is no zero digit, and indicate this by setting the flag `done`. The result was created with `arrayWithCapacity:`, so it is autoreleased.

```
- (void)dealloc
{
   [array release];
   NSZoneFree(NSDefaultMallocZone(), binary);
   [super dealloc];
}

@end

@interface NSArray (Subsets)

- (NSEnumerator *)subsetEnumerator;
```

```
@end

@implementation NSArray (Subsets)

- (NSEnumerator *)subsetEnumerator
{
   NSEnumerator *en =
      [[SubsetEnumerator alloc] initWithArray:self];
   [en autorelease];
   return en;
}

@end
```

It remains to clean up: free the buffer that holds the binary digits and release the array (this matches the retain from `initWithArray`.) We put the method `subsetEnumerator` into a category. The subset enumerator is autoreleased.

You may find yourself wanting to sort subsets according to the number of elements if you use this category. This is done with

```
int acountcmp(id a1, id a2, void *context)
{
   int v1 = [a1 count];
   int v2 = [a2 count];
   if (v1 < v2)
      return NSOrderedAscending;
   else if (v1 > v2)
      return NSOrderedDescending;
   else
      return NSOrderedSame;
}
```

and

```
   en = [args subsetEnumerator];
   all = [[en allObjects]
           sortedArrayUsingFunction:acountcmp
           context:NULL];
```

### 1.1.3 Remark

We can dispense with the array of digits if the number of subsets fits into an unsigned long integer. This simplifies the algorithm quite a lot, since we need only convert the counter value to binary.

The function `subset` extracts binary digits one by one and includes an element if the corresponding digit is one. The elements of the subset are collected at the front of the array, and the remaining locations are marked with a NULL pointer.

```
#include <stdio.h>
#include <string.h>

void subset(char **subs, unsigned long p, int n)
{
   int pos, dest = 0;

   for(pos=0; pos<n; pos++){
      int s = p % 2;
      p = (p-s)/2;
```

```
      if(s){
         subs[dest++] = subs[pos];
      }
   }

   while(dest<n){
      subs[dest++] = NULL;
   }
}
```

The loop at the heart of the program counts from zero to $2^n - 1$.

```
int main(int argc, char** argv, char **env)
{
   if(argc<2){
      printf("need␣at␣least␣one␣item␣to␣form␣set\n");
      exit(1);
   }

   int n=argc-1;
   char **subs = (char **)malloc(n*sizeof(char *));

   unsigned long p, mx = 1<<n;
   for(p=0; p<mx; p++){
      memcpy(subs, argv+1, n*sizeof(char *));
      subset(subs, p, n);

      putchar('[');
      int pos;
      for(pos=0; pos<n; pos++){
         if(subs[pos]==NULL){
            break;
         }
         if(pos){
            putchar('␣');
         }
         printf("%s", subs[pos]);
      }
      puts("]");
   }

   free(subs);

   exit(0);
}
```

## 1.2 Counting II: Permutation enumerator

*by Marko Riedel*

### 1.2.1 Idea

Suppose you wish to permute $n$ items that are stored in an array. You select the last item, then the next-to-last etc. (You could also start from the beginning, going first, second, third etc.) You have $n$ choices for the last item. Then you have $n-1$ choices for the next-to-last item etc. Every permutation corresponds to a vector of choices. If we can enumerate these vectors, then we can enumerate permutations.

A sequence of integers up to some limit is easily enumerated by starting at zero and adding one until we get to the limit. Observe the process in binary: $0, 1, 10, 11, 100, 101, 110, 111$ etc. The weights of the digits are $1, 2, 4, 8$ etc., starting from the right. Now consider a base where the weights are $1, 2, 6, 24, 120$. This is the so-called factorial base (a Cantor base). Individual digits range from 0 to $w_2/w_1 - 1$, where $w_1$ is the weight at that position and $w_2$ the weight at the next position. By the same counting process as in binary, we get $0, 1, 10, 11, 20, 21, 100, 101$ etc. Addition in this base is the same as regular addition. To add one, start at the right and increment going to the left until you get a digit that is not the maximum for that position where maximal digits are replaced by zero. E.g. the last digit of 11 is maximal in its position while the second is not, so $(1)(1) + 1$ yields $(2)(0)$.

We can enumerate permutations by counting from 0 to $n! - 1$ in the factorial base. The first digit indicates how to choose the last element, the second the next-to-last element and so on.

### 1.2.2 Implementation

We will implement an enumerator for permutations and add a category to `NSArray`.

```
@interface PermutationEnumerator : NSEnumerator
{
   NSArray *array;
   int *swapLocs;
   int count;
   BOOL done;
}

- (id)initWithArray:(NSArray *)anArray;
- (id)nextObject;
- (void)dealloc;

@end
```

The state of the enumerator is defined by the array whose elements are being permuted, the number of elements in the array, the counter whose factorial base digits we store in the array `swapLocs` and a flag that indicates whether we are done.

```
@implementation PermutationEnumerator

- (id)initWithArray:(NSArray *)anArray
{
   int index;

   [super init];

   if(!(count = [anArray count])){
      [NSException raise:NSInvalidArgumentException
                  format:@"can't permute zero items"];
   }
   array = anArray; [array retain];

   swapLocs = NSZoneMalloc(NSDefaultMallocZone(),
                           count*sizeof(int));
   for(index=0; index<count; index++){
      swapLocs[index] = 0;
   }

   done = NO;

   return self;
```

```
}
```

The initializer raises an exception if the array is empty. Otherwise the array is retained by the enumerator. We allocate the array for the individual digits in the next call. (Note that we could have used an integer and extracted the individual digits as necessary, by a process that is precisely like converting an integer to binary. Unfortunately this would limit our enumerator to arrays with at most $m$ items, where $m$ is the largest value such that $m!$ can be represented in a machine integer.) We start counting at zero, and hence all the digits are set to zero.

```
- (id)nextObject
{
   NSMutableArray *result;

   int index;
   if(done==YES){
      return nil;
   }

   result = [NSMutableArray arrayWithArray:array];
   for(index=0; index<count; index++){
      int upper = count-1-index;
      if(swapLocs[index]!=upper){
         [result exchangeObjectAtIndex:swapLocs[index]
               withObjectAtIndex:upper];
      }
   }

   for(index=count-1; index>=0; index--){
      if(swapLocs[index]<count-1-index){
         swapLocs[index]++;
         while(++index<count){
            swapLocs[index] = 0;
         }
         break;
      }
   }
   if(index<0){
      done = YES;
   }

   return result;
}
```

The method `nextObject:` constructs the permutation that corresponds to the current value of the counter and increments it. The first digit tells us which of the $n$ items to choose for the last position, the second digit which to choose for the next-to-last position and so on. We increment the counter by scanning digits from the right until we find a digit that is not maximal, and increment it. Digits to the left of this digit are set to zero. We are done if all digits are maximal. The result was created with `arrayWithArray:`, so it is autoreleased.

```
- (void)dealloc
{
   [array release];
   NSZoneFree(NSDefaultMallocZone(), swapLocs);
   [super dealloc];
}
```

```
@end

@interface NSArray (Permute)

- (NSEnumerator *)permutationEnumerator;

@end

@implementation NSArray (Permute)

- (NSEnumerator *)permutationEnumerator
{
   NSEnumerator *en =
      [[PermutationEnumerator alloc] initWithArray:self];
   [en autorelease];
   return en;
}

@end
```

The remaining code assures that the buffer for the digits is freed properly when the enumerator is deallocated and adds a method to `NSArray` that returns an autoreleased permutation enumerator for the contents of the array. This algorithm is based on the random-permutation algorithm from *The Art of Computer Programming.*

### 1.2.3 Remark

We can dispense with the array of digits if the number of permutations fits into an unsigned long integer. This simplifies the algorithm quite a lot, since we need only convert the counter value into the factorial base.

The function `permute` extracts factorial base digits one by one and swaps the appropriate element to the current end of the permutation, where it will remain. The end moves from the top of the array to the bottom.

```
#include <stdio.h>
#include <string.h>

unsigned long factorial(n)
{
   return (n <= 1 ? 1 : n*factorial(n-1));
}

void permute(char **perm, unsigned long p, int n)
{
   int pos;

   for(pos=n; pos>0; pos--){
      int s = p % pos;
      p = (p-s)/pos;

      if(s<pos){
         char *temp;

         temp = perm[pos-1];
         perm[pos-1] = perm[s];
         perm[s] = temp;
      }
   }
```

```
}
```

The loop at the heart of the program counts from zero to $n! - 1$.

```
int main(int argc, char** argv, char **env)
{
   if(argc<2){
      printf("need at least one item to permute\n");
      exit(1);
   }

   int n=argc-1;
   char **perm = (char **)malloc(n*sizeof(char *));

   unsigned long p, mx = factorial(n);
   for(p=0; p<mx; p++){
      memcpy(perm, argv+1, n*sizeof(char *));
      permute(perm, p, n);

      int pos;
      printf("%s", perm[0]);
      for(pos=1; pos<n; pos++){
         printf(" %s", perm[pos]);
      }
      putchar('\n');
   }

   free(perm);

   exit(0);
}
```

## 1.3 Steganography

*by Marko Riedel*

### 1.3.1 Idea

We provide a sample implementation of a basic steganography algorithm, by which hidden data are placed in a TIFF image, which is assumed to contain RGB values in either a planar or a meshed configuration. Think of the input data as a stream of bits. We place these bits in the image by altering the least significant bit of the red color component. The actual location of the data in the image is determined by a passphrase. This assures that even if someone can determine which pixels have been altered, she will not know in what order the altered pixels should be read.

### 1.3.2 Implementation

The first task is to construct a permutation of the pixel locations from the passphrase. We may think of the passphrase as an infinite sequence of bits (repeat the passphrase to get additional bits). We start by implementing a class that delivers chunks of bits from this source.

```
@interface BitSource : NSObject
{
   unsigned char *source;
   int pos, total, len;
}
```

```
+ (NSString *)bitString:(unsigned long)val count:(int)bc;

- initWithString:(NSString *)str;
- (unsigned long)getBits:(int)count;

- (void)dealloc;

@end
```

The class `BitSource` stores the passphrase, which we restrict to ASCII characters in the range from 32 to 127, so that every character provides seven bits. The total number of bits is seven times the length of the passphrase and repeats thereafter. `Bitsource` stores the position in the bit stream, the length of the phrase and the total number of bits. The method `bitString:count:` is used for debugging purposes. It converts some number of bits at the least significant end of an unsigned long integer into a string.

```
@implementation BitSource

+ (NSString *)bitString:(unsigned long)val count:(int)bc
{
   NSString *result = @"";
   int cur;

   for(cur=0; cur<bc; cur++){
      result =
         [(val & 1 ? @"1" : @"0")
            stringByAppendingString:result];

      val >>=1;
   }

   return result;
}
```

We extract the bits one after the other and prepend them to the result string (we prepend because the less significant bits come first).

The method `initWithString:` initializes a bit source from a passphrase. It checks that the phrase is not empty, allocates a buffer for the phrase, checks that no characters are out of range and copies the characters into the buffer. It initializes the postion to be at the beginning of the string and computes the total number of bits that the phrase can provide before it repeats.

```
- initWithString:(NSString *)str
{
   const char *cstr = [str cString], *cur;
   unsigned char *scur;

   [super init];

   len = [str length];
   if(len<1){
      [NSException raise:NSInvalidArgumentException
                  format:@"bit_source_from_empty_string"];
   }


   source = scur = NSZoneMalloc(NSDefaultMallocZone(),
                           len*sizeof(unsigned char));
   for(cur=cstr; *cur; cur++, scur++){
```

```
        unsigned char item = *cur;
        if(item<32 || item>127){
            [NSException raise:NSRangeException
                        format:@"char out of range (32-127): %d",
                        item];
        }
        *scur = item;
    }

    pos = 0; total = len*7;

    return self;
}
```

The method `getBits:` does the actual work. It returns the requested number of bits in an unsigned long integer, scanning the phrase from left to right and wrapping at the end of the phrase.

```
- (unsigned long)getBits:(int)count
{
    unsigned long result = 0;
    int index;

    for(index=0; index<count; index++){
        int whatChar = pos/7, whatBit = 6-(pos%7);
        int bit =
            (source[whatChar] &
             (1 << whatBit)) >> whatBit;

        result = (result << 1) + bit;

        pos = (pos+1)%total;
    }

    return result;
}
```

The method computes what character corresponds to the current position and what bit to extract. It starts with the highest bit because the phrase is scanned from left to right. It writes the bit into the result integer and moves to the next position. It starts over if it reaches the end of the phrase. The method `dealloc` frees the buffer where the phrase was stored.

```
- (void)dealloc
{
    NSZoneFree(NSDefaultMallocZone(), source);
    [super dealloc];
}

@end
```

We can now move on to the actual algorithm. There are several steps.

- Parse command line arguments. These are: `-d/-e`, which tells whether to encode or decode; the passphrase; and the file that contains the TIFF image.
- Use the passphrase to construct a permutation of the possible pixel locations.
- Read or write the length of the secret message.
- Read or write the message itself.

```
#define BUFSIZE 4096

int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];
   NSProcessInfo *procInfo = [NSProcessInfo processInfo];
   NSArray *args = [procInfo arguments];

   NSFileManager *fm = [NSFileManager defaultManager];

   BOOL encode;
   NSString *eswitch, *passphrase, *infile, *outfile;

   BitSource *bits;

   NSImage *img;
   NSEnumerator *repEn;
   NSBitmapImageRep *rep;
   NSSize imgSize;
   unsigned char *dplanes[5];
   int spp;
   BOOL isPlanar;

   unsigned long *permutation;
   int pbits;
   unsigned long width, height, pixels, pindex;

   unsigned char dlbytes[sizeof(unsigned long)];
   long dindex, dlen;
   int bpos, bit;
   unsigned long where;
```

We start with several blocks of declarations: the first block lets us access the command line arguments; the second, the file manager, which we need to check whether the image is readable; the third, various strings, one for each command line argument; the fourth, the bit source; the fifth, the image that we'll manipulate, the bitmap representation that wraps the image data, the data themselves, a flag to tell whether the image is planar or meshed, and how many samples there are per pixel; the sixth, the permutation of the pixel locations, how many bits we'll need for each item in the permutation, and the dimensions of the image; the seventh and last block, a buffer for accessing individual bytes of the long integer that holds the message length and various integers that store state while we process the image.

Processing command line arguments is the easy part. First check that there is the right number of arguments; then check the value of the encode-decode switch.

```
   if([args count]!=4){
      [NSException raise:NSInvalidArgumentException
                  format:@"args are -d/-e passphrase image"];
   }

   eswitch = [args objectAtIndex:1];
   if([eswitch isEqualToString:@"-e"]){
      encode = YES;
   }
   else if([eswitch isEqualToString:@"-d"]){
      encode = NO;
   }
   else{
```

```
    [NSException raise:NSInvalidArgumentException
             format:@"-d/-e (de/encode) expected"];
}
```

Initialize the bit source from the passphrase:

```
passphrase = [args objectAtIndex:2];
bits =
   [[BitSource alloc] initWithString:passphrase];
```

Get the file name and check that the file is readable, and construct the name of the output file, which we obtain by placing the string `-stg` between the extension `TIFF` and the rest of the filename.

```
infile = [args objectAtIndex:3];
if([fm isReadableFileAtPath:infile]==NO){
   [NSException raise:NSInvalidArgumentException
             format:@"%@ isn't readable", infile];
}

outfile =
   [[[infile stringByDeletingPathExtension]
       stringByAppendingString:@"-stg"]
      stringByAppendingPathExtension:@"tiff"];
```

Now open the image, extract the image dimensions and compute the total number of pixels. Compute the integer logarithm of this number. It tells us how many bits we need to request from the bit source when we construct the permutation of pixel locations.

```
img = [[NSImage alloc] initWithContentsOfFile:infile];
if(img==nil){
   [NSException raise:NSInvalidArgumentException
             format:@"no image in %@", infile];
}

imgSize = [img size];
width = imgSize.width;
height = imgSize.height;
pixels = width*height;

pbits = 1;
while(pixels > (1<<pbits)){
   pbits++;
}
```

The next step is to actually construct the permutation. We use the idea from *The Art of Computer Programming*, also discussed elsewhere in this document. Start with a fully ordered permutation and move an item it to the end; decrease the end marker, repeat. The choice of the item is not random in our case. It is determined by reading the required number of bits from the bit source. The item that we move to the end is given by the remainder of the bits modulo the length from the start to the current end marker.

```
permutation = NSZoneMalloc(NSDefaultMallocZone(),
                    pixels*sizeof(unsigned long));
if(permutation==NULL){
   [NSException raise:NSMallocException
             format:@"malloc failed (%ld)", pixels];
}

for(pindex=0; pindex<pixels; pindex++){
   permutation[pindex] = pindex;
```

```
}
NSLog(@"width %ld height %ld pixels %ld bits %d",
     width, height, pixels, pbits);

for(pindex=pixels-1; pindex>0; pindex--){
   int swaploc = [bits getBits:pbits] % (pindex+1);
   unsigned long item;

   item            = permutation[pindex];
   permutation[pindex] = permutation[swaploc];
   permutation[swaploc] = item;
}
```

The last step before the actual computation is to gain access to the data planes where the image is stored. The red data plane comes first in a planar configuration. We obtain the data by looking for a bitmap representation among the representations of the image and exit if no such representation is found. Otherwise we ask the representation for the data, and set the planarity flag and the number of samples per pixel.

```
repEn = [[img representations] objectEnumerator];
while((rep = [repEn nextObject])!=nil){
   if([rep isKindOfClass:[NSBitmapImageRep class]]==YES){
      break;
   }
}
if(rep==nil){
   [NSException raise:NSInvalidArgumentException
            format:@"no bitmap image rep for %@", infile];
}

[rep getBitmapDataPlanes:dplanes];
spp = [rep samplesPerPixel];
isPlanar = [rep isPlanar];
```

We now discuss the encode/decode process. The first step in the encode process is to compute the number of bytes the image can hold, read the data from the standard input and check that they fit into the image. We need an extra `sizeof(unsigned long)` bytes to store the length of the message.

```
if(encode==YES){
   NSData *data;
   unsigned long dmax = pixels/8-sizeof(unsigned long);
   const unsigned char *dbytes;

   data = [[NSFileHandle fileHandleWithStandardInput]
           readDataToEndOfFile];
   if((dlen = [data length])>dmax){
      [NSException raise:NSInvalidArgumentException
                format:@"image can only hold %ld bytes",
                dmax];
   }
   else if(!dlen){
      [NSException raise:NSInvalidArgumentException
                format:@"no data"];
   }
```

The actual write process consists of two steps: first write the message length, then write the message. The variable `dlbytes` provides access to the individual bytes of the message length value; `dbytes`, to the bytes of the bitmap data plane. We iterate over a range from negative the number of bytes that hold the

message length, to the length of the message. First extract the byte that is about to be written. Then write all eight bits, one after the other. The permutation tells us where in the image the bit goes. We write the bit by first zeroing the target bit and then setting the target bit to the bit that we want to record. Note that there is a fifty percent chance in a diverse image that the target bit already agrees with the source bit and no change will be detected. We must multiply the index given by the permutation by the number of samples per pixel if the image is not planar (meshed). We write the most significant bit first to facilitate later reads.

```
*(unsigned long *)dlbytes = dlen;
dbytes = [data bytes];

for(pindex = 0, dindex=-sizeof(unsigned long);
   dindex<dlen; dindex++){
   unsigned char tbyte,
      byte = (dindex<0 ?
            dlbytes[-dindex-1] : dbytes[dindex]);

   for(bpos=0; bpos<8; bpos++){
      where = permutation[pindex++];

      if(isPlanar==NO){
         where *= spp;
      }

      bit = (byte & 1<<(7-bpos) ? 1 : 0);

      tbyte = dplanes[0][where];
      tbyte >>= 1; tbyte <<= 1; tbyte += bit;
      dplanes[0][where] = tbyte;
   }
}
```

The last step of the encoding phase is to write the data to the output file.

```
data = [rep TIFFRepresentation];
if([data writeToFile:outfile atomically:NO]==NO){
   [NSException raise:@"data write failed"
               format:@"couldn't write %@", outfile];
}
```

The decode process is even simpler. Start by allocating a mutable data object for the contents of the secret message. In fact we'll write the message contents into a buffer and append the buffer contents to the data object when the buffer is full, so as not to invoke `appendBytes:length:` for every byte that we have extracted. Recall that the first `sizeof(unsigned long)` bytes contain the message length. We set the upper end of the loop range once this value has been extracted. We extract one byte at a time. The permutation tells us where the individual bits are located. We retrieve the least significant bit and write it to the byte being extracted. Recall that the most significant bit comes first, so we can just shift the byte when we record a new bit. We append the contents of the buffer to the data object whenever the buffer is full.

```
NSMutableData *data =
   [NSMutableData dataWithCapacity:8];
unsigned char buf[BUFSIZE], *bptr = buf;

dlen = 0;
for(pindex = 0, dindex=-sizeof(unsigned long);
   dindex<dlen; dindex++){
   unsigned char sbyte = 0;
```

```
        for(bpos=0; bpos<8; bpos++){
           where = permutation[pindex++];

           if(isPlanar==NO){
              where *= spp;
           }

           bit = dplanes[0][where] & 1;
           sbyte = (sbyte << 1) + bit;
        }

        if(dindex<0){
           dlbytes[-dindex-1] = sbyte;
        }
        else{
           *bptr++ = sbyte;
           if(bptr-buf==BUFSIZE){
              [data appendBytes:buf length:BUFSIZE];
              bptr = buf;
           }
        }

        if(dindex==-1){
           dlen = *(unsigned long *)dlbytes;
           NSLog(@"message length is %ld", dlen);
        }
     }
```

If there are data in the buffer at the end of the loop we record them in the data object. The last step of the decode process is to write the contents of the secret message to the standard output.

```
     if(bptr>buf){
       [data appendBytes:buf length:bptr-buf];
     }

     [(NSFileHandle *)
        [NSFileHandle fileHandleWithStandardOutput]
        writeData:data];
```

The program frees memory that has been allocated before it exits.

```
   NSZoneFree(NSDefaultMallocZone(), permutation);

   [bits release];
   [pool release];
   exit(0);
}
```

## 1.4 Working with GMP integers

*by Marko Riedel*

### 1.4.1 Idea

Combinatorics problems often require arbitrary precision arithmetic; this is the case e.g. with the graph enumeration problem that we present in the next recipe. We need an efficient Objective C wrapper class that encapsulates the GMP integer data structure. This is what `GMPInt` is all about.

### 1.4.2 Implementation

The class `GMPInt` is a wrapper class whose sole instance variable holds a GMP Integer.

```
#include <Foundation/Foundation.h>
#include <gmp.h>

@interface GMPInt : NSObject
{
   mpz_t value;
}
```

The class knows how to manufacture objects for the values zero and one and it can compute the factorial of an `unsigned long int`.

```
+ (GMPInt *)zeroObj;
+ (GMPInt *)oneObj;

+ (GMPInt *)factorialUI:(unsigned long)op;
```

There are two factory methods, which use a signed long integer or a GMP integer.

```
+ (GMPInt *)IntWithSI:(signed long)lval;
+ (GMPInt *)IntWithValue:(mpz_t)val;
```

The basic operations of arithmetic are implemented. All four use the receiver as the first operand, the argument as the second and return a `GMPInt` with the value of the result.

```
- (GMPInt *)add:(GMPInt *)op;
- (GMPInt *)sub:(GMPInt *)op;
- (GMPInt *)mul:(GMPInt *)op;
- (GMPInt *)div:(GMPInt *)op;
```

`GMPInt` instances can raise themselves to a power given by an unsigned long integer.

```
- (GMPInt *)powUI:(unsigned long)op;
```

It is important that a `GMPInt` instance be able to represent itself as a string. This is what `stringValue:` does; we also implement `description` so that the `%@` works e.g. in calls to `NSLog`.

```
- (NSString *)stringValue;
- (NSString *)description;
```

Two commonly used predicates test for the receiver's value being zero or one.

```
- (BOOL)isZero;
- (BOOL)isOne;
```

The arithmetic methods need to get at the GMP integer value of their second argument; we provide a method for this purpose. We also implement `dealloc`, so that we can free GMP integers, i.e. ‘‘`mpz_t`’’ values, that are no longer used.

```
- (mpz_t *)valPtr;

- (void)dealloc;

@end
```

The implementation is straightforward. The methods `zeroObj` and `oneObj` simply invoke the factory method with the appropriate argument.

```
@implementation GMPInt

+ (GMPInt *)zeroObj
{
   return [GMPInt IntWithSI:0];
}

+ (GMPInt *)oneObj
{
   return [GMPInt IntWithSI:1];
}
```

The class method for the factorial of an unsigned long integer allocates a `GMPInt`, obtains a pointer to its value, initializes the value (an `mpz_t`), calls GMP to compute the result and returns the `GMPInt`.

```
+ (GMPInt *)factorialUI:(unsigned long)op
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];

   mpz_init(*rptr);
   mpz_fac_ui(*rptr, op);

   return res;
}
```

The two factory methods initialize and set the instance variable `value` from their arguments and put the object into an autorelease pool. This is important, because a computation may temporarily allocate a lot of integers and the space they occupy will not be released if they are not put in an autorelease pool. We'll see more about garbage collection later on.

```
+ (GMPInt *)IntWithSI:(signed long)lval
{
   GMPInt *inst = [GMPInt alloc];
   mpz_t *ptr = [inst valPtr];

   mpz_init_set_si(*ptr, lval);

   AUTORELEASE(inst);
   return inst;
}

+ (GMPInt *)IntWithValue:(mpz_t)val
{
   GMPInt *inst = [GMPInt alloc];
   mpz_t *ptr = [inst valPtr];

   mpz_set(*ptr, val);

   AUTORELEASE(inst);
   return inst;
}
```

The method `add` allocates a `GMPInt`, initializes its value, stores the sum in this value and returns the `GMPInt`.

```
- (GMPInt *)add:(GMPInt *)op
```

```
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];

   mpz_init(*rptr);
   mpz_add(*rptr, value, *[op valPtr]);

   return res;
}
```

Subtraction only differs in the `mpz` method that is called.

```
- (GMPInt *)sub:(GMPInt *)op
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];

   mpz_init(*rptr);
   mpz_sub(*rptr, value, *[op valPtr]);

   return res;
}
```

The structure stays the same for multiplication.

```
- (GMPInt *)mul:(GMPInt *)op
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];

   mpz_init(*rptr);
   mpz_mul(*rptr, value, *[op valPtr]);

   return res;
}
```

Division is like multiplication, except that we must choose from several `mpz` calls according to the rounding style that is to be used and whether we need the quotient, the remainder or both. We choose "truncate" for our rounding style and we do not compute the remainder.

```
- (GMPInt *)div:(GMPInt *)op
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];

   mpz_init(*rptr);
   mpz_tdiv_q(*rptr, value, *[op valPtr]);

   return res;
}
```

Exponentiation differs from the preceding methods in that there is no invocation of `valPtr`, because we may use the argument as is.

```
- (GMPInt *)powUI:(unsigned long)op
{
   GMPInt *res = [[GMPInt alloc] autorelease];
   mpz_t *rptr = [res valPtr];
```

```
    mpz_init(*rptr);
    mpz_pow_ui(*rptr, value, op);

    return res;
}
```

The method `stringValue` returns a string that is a decimal representation of the receiver. It uses `mpz_sizeinbase` to determine the number of digits and takes a possible sign and the terminating null byte into account. It invokes `mpgz_get_str` to write the decimal representation into a buffer it has allocated for this purpose and then produces a `NSString` object from the contents of the buffer, freeing it after the string object is obtained. The method `description` simply returns the string value of the receiver.

```
- (NSString *)stringValue
{
    NSZone *dz = NSDefaultMallocZone();
    char *buf;
    NSString *res;

    buf = NSZoneMalloc(dz, mpz_sizeinbase(value, 10)+2);
    mpz_get_str(buf, 10, value);
    res = [NSString stringWithCString:buf];
    NSZoneFree(dz, buf);

    return res;
}

- (NSString *)description
{
    return [self stringValue];
}
```

The predicates that test for zero and one, respectively, work by calling the `mpz_comp_ui` with the appropriate arguments.

```
- (BOOL)isZero
{
    return (!mpz_cmp_ui(value, 0) ? YES : NO);
}

- (BOOL)isOne
{
    return (!mpz_cmp_ui(value, 1) ? YES : NO);
}
```

The method `valPtr` is used by the arithmetic methods and returns the address of the `mpz_t` value in the receiver.

```
- (mpz_t *)valPtr
{
    return &value;
}
```

The implementation of `GMPInt` concludes with the method `dealloc`, whose purpose is to free the space that GMP has allocated for the value of the receiver, since it is no longer needed.

```
- (void)dealloc
{
    mpz_clear(value);
    [super dealloc];
```

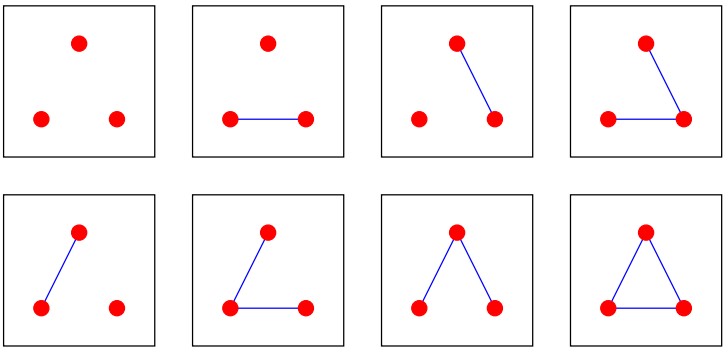

Figure 1: All eight graphs on three vertices.

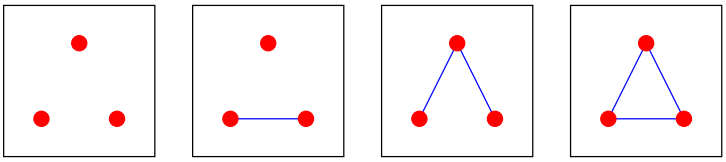

Figure 2: All non-isomorphic graphs on three vertices.

```
}

@end
```

## 1.5 Enumerating graphs with Pólya's theorem and GMP

*by Marko Riedel*

### 1.5.1 Introduction

The object of this recipe is to enumerate non-isomorphic graphs on $n$ vertices using Pólya's theorem and GMP (the GNU multiple precision arithmetic library). We will be concerned with the algorithmics rather than the mathematics. You are encouraged to learn more about the latter, e.g. from Frank Harary's books on the subject.

Fig. 1 shows all possible graphs on three vertices. There are eight of these; for a graph on $n$ vertices we have $2^{n(n-1)/2}$ such graphs. We want a different statistic, namely one that takes isomorphisms into account. Fig. 2 shows that there are really only four different graphs on three vertices. This is because e.g. there exists a vertex permutation that maps the three graphs with one edge to one another. We say that these three graphs constitute an equivalence class.

Let us now reformulate the problem so that Pólya's theorem applies. Given $n$, we want to know how many non-isomorphic graphs, i.e. equivalence classes there are with $k$ edges. We can think of the edges as slots that are either filled (edge present) or empty (edge absent). An empty slot has size zero and a filled one, size one. The number of edges is the sum of the sizes of the slots. The slots (edge locations) are being permuted by any of the $n!$ vertex permutations. There is one edge permutation for each vertex permutation. We will obtain the values for all $k$ at once, namely in a generating function, i.e. in a function where the term $qz^k$ denotes the presence of $q$ non-isomorphic graphs with $k$ edges.

The generic definition of our problem is this: given a set of slots that are being permuted by a permutation group (more on this later) and a set of objects of different sizes that go into the slots (one object per slot), count the number of equivalence classes of filled slot configurations with respect to the permutation group whose total size (i.e. sum of the sizes of the objects placed in the slots) is $k$. In our case the group is the pair group obtained from the symmetric group acting on $n$ vertices and the objects that go into the slots are present/absent edges.

*Example.* Recall Fig. 1 and label the vertices 1, 2, 3. The table shows the vertex and the corresponding edge permutations (the three edges are 12, 13 and 23; the permutations are in table notation, e.g. $1 \to 2$, $2 \to 3$ and $3 \to 1$ is written as 231):

| | |
|---|---|
| 1 2 3 | 12 13 23 |
| 1 3 2 | 13 12 23 |
| 2 1 3 | 12 23 13 |
| 2 3 1 | 23 12 13 |
| 3 1 2 | 13 23 12 |
| 3 2 1 | 23 13 12 |

The eight graphs correspond to the three slots (edges) being filled or empty.

| 12 | 13 | 23 |
|---|---|---|
| empty | empty | empty |
| empty | empty | filled |
| empty | filled | empty |
| empty | filled | filled |
| filled | empty | empty |
| filled | empty | filled |
| filled | filled | empty |
| filled | filled | filled |

The edge permutations permute the columns of the table, e.g. 12 23 13 takes line two to line three, so these are in the same equivalence class. Permuting edges does not change the total size of the configuration (number of edges), so there is no permutation that takes line two to line four.

Another common application of Pólya's theorem is to count, say, the number of different necklaces, where a necklace is made up of $n$ gemstones and there are $m$ different types of gemstones (colors) available. We can choose these to have the same size or different sizes. In terms of the above discussion, the slots are the positions on the necklace, the objects that go into the slots are the gemstones and the permutations rotate or flip the necklace or both.

### 1.5.2 Permutations

We will be using only a few basic facts about permutations and permutation groups. The elements of the latter are permutations and the multiplication operation is permutation composition. There must not be any permutations whose product is not in the group. E.g. $\{123, 231\}$ is not a group because $231 \circ 231 = 312$, which is not in the set. We are concerned with two groups: the symmetric group $S_n$ on $n$ vertices, which includes all $n!$ permutations and the corresponding pair group $S_n^{(2)}$ on $n(n-1)/2$ edges, whose elements are obtained from the action of the vertex permutation on the edges. E.g. consider the vertex permutation 23451 and the edges 12 13 14 15 23 24 25 34 35 45. The corresponding edge permutation $\epsilon$ is 23 24 25 12 34 35 13 45 14 15.

We now switch to a more useful way of writing permutations, i.e. the so-called disjoint cycle decomposition. This consists in writing the permutation as a product of its cycles. E.g. the permutation 164325 (table notation) would be written as (1) (265) (34) since e.g. it takes 2 to 6, 6 to 5 and 5 back to 2, and the edge permutation $\epsilon$ as (12 23 34 45 15) (13 24 35 14 25).

The disjoint cycle decomposition tells us about the shape of the permutation. This shape information is captured by a product notation, where the variable $a_k$ stands for a cycle of length $k$ and $j_k$ for the number of cycles with this length, giving the product $\prod_{k=1}^{p} a_k^{j_k}$. E.g. the product for (1) (265) (34) would be $a_1 a_2 a_3$, whith $j_1 = j_2 = j_3 = 1$ and $j_4 = j_5 = j_6 = 0$ and for $\epsilon$, $a_5^2$, with $j_1 = j_2 = j_3 = j_4 = 0$, $j_5 = 2$ and $j_6 = j_7 = j_8 = j_9 = j_{10} = 0$.

### 1.5.3 The Cycle Index

The cycle index $Z(S)$ of a permutation group $S$ is the average of $\prod_{k=1}^{p} a_k^{j_k}$ over all permutations in the group. E.g. the cycle indices of the symmetric group $S_3$ and the pair group $S_3^{(2)}$ are obtained from the following table.

| | | | |
|---|---|---|---|
| 1 2 3 | $a_1^3$ | 12 13 23 | $a_1^3$ |
| 1 3 2 | $a_1a_2$ | 13 12 23 | $a_1a_2$ |
| 2 1 3 | $a_1a_2$ | 12 23 13 | $a_1a_2$ |
| 2 3 1 | $a_3$ | 23 12 13 | $a_3$ |
| 3 1 2 | $a_3$ | 13 23 12 | $a_3$ |
| 3 2 1 | $a_1a_2$ | 23 13 12 | $a_1a_2$ |

This special case yields

$$Z(S_3) = Z(S_3^{(2)}) = \frac{1}{6}a_1^3 + \frac{1}{2}a_1a_2 + \frac{1}{3}a_3.$$

### 1.5.4 Pólya's theorem

This is what the theorem says: suppose you have some set of objects of different sizes and distribute these objects into $n$ slots, where a permutation group acts on the slots to create equivalence classes of filled slot configurations. The generating function of these equivalence classes is obtained by replacing $a_k$ by $f(z^k)$ in the cycle index, where $f$ is the generating function of the objects.

The permutation group is $S_n^{(2)}$ (pair group acting on the vertices) when we enumerate graphs and the generating function of the objects is $1 + z$, indicating whether an edge is present (size one, $z$) or not (1). Substitute $a_k = 1 + z^k$ into $Z(S_3^{(2)})$ to obtain

$$\frac{1}{6}(1+z)^3+\frac{1}{2}(1+z)(1+z^2)+\frac{1}{3}(1+z^3) = \frac{1}{6}z^3+\frac{1}{2}z^2+\frac{1}{2}z+\frac{1}{6}+\frac{1}{2}z^3+\frac{1}{2}z^2+\frac{1}{2}z+\frac{1}{2}+\frac{1}{3}z^3+\frac{1}{3} = z^3+z^2+z+1,$$

which says that there is one graph with three edges, one with two, one with one edge and one with no edges, as is also evident from Fig.2.

### 1.5.5 Cycle index of the pair group

We need the cyle index of $S_n^{(2)}$ in order to enumerate graphs. There is one permutation for every permutation from $S_n$ in $S_n^{(2)}$. Fortunately we needn't iterate over all $n!$ permutations in $S_n$, which would make the computation infeasible for all but small $n$. Recall that the shape of the permutation determines the product that it contributes to the cycle index. This means that e.g. (1 2 3) (4) (5 6), (1) (2 3) (4 5 6) and (1 2) (3 4 5) (6) all contribute the same term to $S_6$, namely $a_1a_2a_3$ (three copies of it).

Hence we can enumerate the permutations in $S_n$ by iterating over all possible shapes and computing how often each shape occurs. These shapes are precisely the partitions of $n$ into some number of summands. E.g. the cycle index for $S_3$ contains one product term for each of the possible partitions $1+1+1$, $1+2$ and 3. We need the coefficient associated to each of these three partitions, i.e. the number of times this shape occurs.

The disjoint cycle decomposition $\prod_{k=1}^n a_k^{j_k}$ occurs

$$\frac{n!}{\prod_{k=1}^n (k!)^{j_k}} \prod_{k=1}^n \left(\frac{k!}{k}\right)^{j_k} \prod_{k=1}^n \frac{1}{j_k!} = \frac{n!}{\prod_{k=1}^n k^{j_k} j_k!}$$

times. (Partition the $n$ elements into subsets, one for each cycle. A subset of size $k$ generates $k!/k$ cycles. The same size $j_k$ set of $k$-cycles may be chosen in $j_k!$ different ways.)

For $S_3$ this yields the following results:

- $1+1+1$: choose three subsets of size one in $3!/1!/1!/1! = 6$ ways, every such subset generates $1!/1 = 1$ cycles but all 3! choices for the three subsets are the same, giving 1.
- $1 + 2$: choose a subset of size two and one of size one in $3!/1!/2! = 3$ ways (there are three ways to choose the singleton): the size two subset generates $2!/2 = 1$ cycle and the size one subset, $1!/1 = 1$ cycle; there no duplicates in the choice of size 2 or size 1 subsets, giving 3.
- 3: choose a subset of size three in $3!/3! = 1$ ways, this subset generates $3!/3 = 2$ cycles and again there are no duplicates in the choice of the size 3 subset, giving 2.

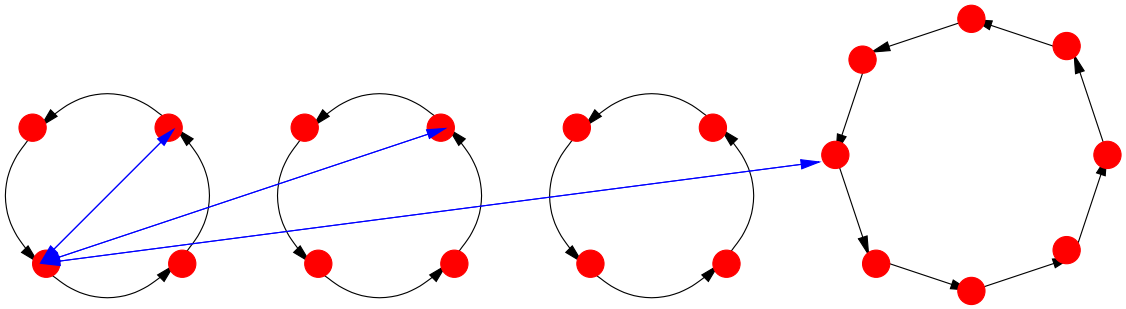

Figure 3: Vertex permutation with the shape $a_4^3\, a_8$ and the three possible types of edges.

This computation confirms the value for $S_3$ that we computed earlier.

How to obtain the possible partitions? This is simple, since there is a folklore algorithm for the problem. Instead of computing the set of partitions $P_n$ of $n$, we start with $P_{n,k}$, the set of partitions of $n$ into $k$ summands. Now every partition in $P_{n,k}$ either contains one, and can be obtained by including one in a partition from $P(n-1,k-1)$, or it does not contain one and $n-k >= k$, and it can be obtained from a partition from $P(n-k,k)$ by incrementing its elements. We can compute $P_{n,k}$ recursively, the base case being $P_{1,1} = \{[1]\}$. $P_n$ is the union of the $P_{n,k}$ for $1 \leq k \leq n$.

At this point we have all possible shapes and we know how often they occur. The last step is to compute the shape of the edge permutation from the shape of the vertex permutation. An edge connects two vertices. There are three possiblities: these vertices lie on the same cycle of the vertex permutation, they lie on different cycles of the same length or they lie on different cycles of different lengths. These three possibilities are shown in Fig. 3. We must compute the length of the edge cycle (i.e. how long it takes for the vertex pair to return to the start edge when it moves in parallel along the vertex cycle) in each case and how often it occurs. There is an additional distinction to be made when the vertices lie on the same cycle: a cycle of even length contains vertex pairs that return to the start edge after having covered exactly half, rather than the whole cycle. Furthermore, the number of steps for the return to the start position for two vertices on cycles of different lengths is given by the least common multiple of those lengths. Note that we'll overcount cycles by a factor that is equal to the length of the edge cycle. We illustrate with a sample computation.

Say the vertex permutation has the shape $a_3^2\, a_6\, a_7$.

- **Same cycle.**
  - $a_3^2$: three vertex pairs, length three, twice: $a_3^2$
  - $a_6$: fifteen vertex pairs, three short cycles: $a_6^2 a_3$
  - $a_7$: twenty-one vertex pairs, length seven, $a_7^3$
- **Different cycles, same length.**
  - $a_3^2$: nine vertex pairs, length three: $a_3^3$
- **Different cycles, different length.**
  - $a_3^2 a_6$: eighteen vertex pairs, length six, twice, $a_6^6$
  - $a_3^2 a_7$: twenty-one vertex pairs, length twenty-one, twice, $a_{21}^2$
  - $a_6 a_7$: forty-two vertex pairs, length forty-two: $a_{42}$.

This shows that the corresponding edge permutation has the shape $a_3^6\, a_6^8\, a_7^3\, a_{21}^2\, a_{42}$, and indeed $18 + 48 + 21 + 42 + 42 = 171 = 1/2\ 19\ (19-1)$.

We now have all the ingredients to implement the graph enumeration algorithm. There will be four steps:

- Compute the partitions of $n$.

- Compute the vertex permutation shapes from the partitions and the coefficient.
- Compute the edge permutation shape, substitute $1+z$ and add the resulting polynomial to the current total.
- Divide by $n!$ to obtain the generating function.

Note that we average the cycle index in the last step in order to be able to work with integers during the whole computation. Speaking of integers, our implementation will use the GMPInt wrapper class for GMP integers.

### 1.5.6 Concerning memory allocation

The computation that we undertake produces many intermediate results. These are: `GMPInt` values to represent coefficients of the products whose sum forms the cycle index; `NSString` objects that document the computation if the user activates debugging and `NSMutableArray` objects that hold partitions and product terms as well as polynomials in one variable that result from the substitution of $1 + z$ into the cycle index. All of these can use up a lot of memory very quickly, which is why we absolutely must implement garbage collection. This is done in the loop that iterates over the partitions and hence over the products that occur in the cycle index. We check whether the number of allocated critical objects has doubled, and empty the autorelease pool if it has.

### 1.5.7 Implementation I: auxiliary functions

There is a preliminary remark concerning our terminology. We use the term multiset to refer to partitions and products. This is because a partition like $[1, 1, 1, 2, 3]$ or $a_1^3\ a_2\ a_3$ is a multiset containing three copies of the value "1" and one copy each of the values "2" and "3."

Our list of auxiliary functions starts with the basic GCD algorithm. The GCD $g$ of $p$ and $q$ divides $p = mq + r$ and $q$, so it also divides $r < q$. Hence replace $(p, q)$ by $(q, r)$ and repeat until $r = 0$. We need the GCD to compute the LCM of the lengths of two cycles of a vertex permutation. The LCM gives the length of the edge cycle of those edges that have one vertex on each of the two vertex cycles.

```
int gcd(int p, int q)
{
    int r;

    if(p<q){
        int s;

        s = p;
        p = q;
        q = s;
    }

    while(r=p%q){
        p = q;
        q = r;
    }

    return q;
}
```

The function `SumIt` sums a multiset, where each element contributes according to its multiplicity and the first element has value one. It iterates over the elements in a loop. This is very useful for consistency checks. The cycle lengths of a vertex permutation must sum to $n$, and the cycle lengths of the corresponding edge permutation to $1/2n(n-1)$.

```
int SumIt(NSArray *data)
{
```

```
   int sum, pos, max = [data count];

   for(sum=0, pos=0; pos<max; pos++){
      sum += (pos+1)*
         [[data objectAtIndex:pos] intValue];
   }

   return sum;
}
```

The next two functions are very important; they compute $P_{n,k}$ and $P_n$, respectively. Start with the former. The base case comes first: the answer is $\{[1]\}$ when $n = 1$. The data structure that we use is an array of arrays of integer `NSNumber` objects. The number is one summand from the partition, the inner array a single partition and the outer array a collection of partitions.

```
NSMutableArray *Partitionsk(int n, int k)
{
   NSMutableArray *part,
      *result = [NSMutableArray arrayWithCapacity:1];
   NSEnumerator *en;

   if(n==1){
      part =
         [NSMutableArray
            arrayWithObject:
               [NSNumber numberWithInt:1]];
      [result addObject:part];
      return result;
   }
```

The two recursions are next. Either the partition contains one or it doesn't. There is a one-to-one correspondence with a permutation of $P_{n-1,k-1}$ in the former case, so obtain those and collect them, adding a summand with value one at the front of each summand array.

```
   if(k>1){
      en = [Partitionsk(n-1, k-1) objectEnumerator];
      while((part = [en nextObject])!=nil){
         [part insertObject:[NSNumber numberWithInt:1]
             atIndex:0];
         [result addObject:part];
      }
   }
```

If the partition does not contain one and $n - k$ is large enough for $k$ summands, then we have a one-to-one correspondence with a permutation from $P_{n-k,k}$. We obtain these permutations and increment each summand of each partition, collecting the new partitions in the array `result`. These are the only possible cases and we are done.

```
   if(n-k>=k){
      en = [Partitionsk(n-k, k) objectEnumerator];
      while((part = [en nextObject])!=nil){
         NSMutableArray *npart =
            [NSMutableArray arrayWithCapacity:k];
         int pos;
         for(pos=0; pos<k; pos++){
            int val = [[part objectAtIndex:pos] intValue];
            [npart
               addObject:
```

```
                    [NSNumber numberWithInt:val+1]];
         }
         [result addObject:npart];
      }
   }

   return result;
}
```

Obtaining all partitions of $n$ is now easy: compute the union of $P_{n,k}$ for $1 \leq k \leq n$. This is done with a loop.

```
NSMutableArray *Partitions(int n)
{
   int k;
   NSMutableArray *result =
      [NSMutableArray arrayWithCapacity:1];

   for(k=1; k<=n; k++){
      NSMutableArray *partsk = Partitionsk(n, k);
      [result addObjectsFromArray:partsk];
   }

   return result;
}
```

We will need to turn partitions into multisets, i.e. vertex permutation shapes that represent product terms of the form $\prod_{k=1}^{n} a_k^{j_k}$. This product correponds to a partition that contains $j_k$ instances of the value $k$. E.g. we want to turn $[1,1,1,2,2,3]$ into $a_1^3\, a_2^2\, a_3$, which is represented as the array $[3,2,1]$. First initialize the array with zero, i.e. in our example we would have $[0,0,0]$ and the capacity of the array would be three.

```
NSMutableArray *PartitionToMSet(NSMutableArray *part)
{
   int pos, max = [[part lastObject] intValue];
   NSMutableArray *result =
      [NSMutableArray arrayWithCapacity:max];

   for(pos=0; pos<max; pos++){
      [result addObject:[NSNumber numberWithInt:0]];
   }
```

Then iterate over the partion and record a summand with value `value` (recall that these values are wrapped in `NSNumber` objects) at position `value-1` of the multiset array. This is done by reading the old value, incrementing it, and writing it back into the array. This would yield the sequence $[1,0,0]$, $[2,0,0]$, $[3,0,0]$, $[3,1,0]$, $[3,2,0]$, and $[3,2,1]$ in the example.

```
   NSEnumerator *en = [part objectEnumerator];
   NSNumber *val;
   while((val = [en nextObject])!=nil){
      int value = [val intValue],
         count = [[result objectAtIndex:value-1] intValue];
      [result replaceObjectAtIndex:value-1
            withObject:[NSNumber numberWithInt:count+1]];
   }

   return result;
}
```

Our program provides an extensive trace/debug facility that makes it possible for the user to follow the computation very closely. Hence it is imperative that we be able to convert from the internal representation of a multiset into a string representation, i.e. we must be able to turn $[3, 2, 1]$ into the string $a_1^3\ a_2^2\ a_3$. This is done by iterating over the multiset, reading the entry at that position, and storing a string representation of the term in the array `factors`. (Empty fields are skipped.)

```
NSString *MSetToProduct(NSMutableArray *mset)
{
   NSNumber *cval;
   NSMutableArray *factors =
      [NSMutableArray arrayWithCapacity:1];
   int pos, max = [mset count];

   for(pos=0; pos<max; pos++){
      int count = [[mset objectAtIndex:pos] intValue];
```

The string representation of such a term has the form $a_k$ if if $j_k$ is one, and the form $a_k^{j_k}$ otherwise.

```
      if(count){
         NSString *item =
            [NSString stringWithFormat:@"a_%d",
                    pos+1];
         if(count>1){
            item = [item stringByAppendingFormat:@"^%d",
                      count];
         }
         [factors addObject:item];
```

We join all terms that have been collected by concatenating them. (They are separated by a single space in the result string.)

```
      }
   }

   return [factors componentsJoinedByString:@" "];
}
```

Recall that the number of times a shape occurs in the symmetric group is given by the formula

$$\frac{n!}{\prod_{k=1}^{n} k^{j_k} j_k!}.$$

The next procedure computes the denominator of this fraction. Set up a loop to iterate over the multiset and process nonzero entries only.

```
GMPInt *MSetToCoeffDenom(NSMutableArray *mset)
{
   GMPInt *one = [GMPInt oneObj],
      *res = one, *k = one;
   int pos, max = [mset count];

   for(pos=0; pos<max; pos++){
      int count = [[mset objectAtIndex:pos] intValue];
      if(count){
```

The actual computation is to multiply the current denominator by $k^{j_k}$ and $j_k!$.

```
         GMPInt
            *m1 = [k powUI:count],
            *m2 = [GMPInt factorialUI:count];
```

```
        res = [res mul:m1];
        res = [res mul:m2];
    }
```

We increment $k$ at the end of the loop body and return the denominator once the loop exits.

```
    k = [k add:one];
  }

  return res;
}
```

The next function is really more of a macro that we'll use to turn vertex permutation shapes into edge permutation shapes, which is why it's marked `inline`. It records an increase of a product term $a_k$ by some amount.

```
inline void AddToMSet(NSMutableArray *mset, int len, int count)
{
  int val = [[mset objectAtIndex:len-1] intValue];
  [mset replaceObjectAtIndex:len-1
       withObject:[NSNumber numberWithInt:val+count]];
}
```

The next function is of critical importance: it produces the terms of the cycle index of the pair group. The maximum possible length of an edge cycle obtained from a vertex permutation is $l(l-1)$, where $l$ is the length of the longest cycle of the vertex permutation. This occurs if there are cycles of length $l-1$ and applies to edges with one vertex on the $l-1$-cycle and the other on the $l$-cycle. The maximum length is one for the identity permutation. The multiset of edge cycles is initialized to be empty.

```
NSMutableArray *VertexMSetToEdgeMSet(NSMutableArray *vmset)
{
  int vmxlen = [vmset count], vlen1, vlen2,
    upper = 0,
    emxlen = (vmxlen==1 ? 1 : vmxlen*(vmxlen-1)), elen;
  int inst1, inst2, instcount;
  NSMutableArray *emset =
    [NSMutableArray arrayWithCapacity:emxlen];

  for(elen=1; elen<=emxlen; elen++){
    [emset addObject:[NSNumber numberWithInt:0]];
  }
```

The first case occurs when there are two vertices on the same cycle. If the cycle length is odd, then the edge cycle has the same length and the number of edges on such cycles is the number of different vertex pairs on the cycle. We scale (divide) by the length of the cycle because each cycle is counted once for each edge on it. We must also scale the term by its multiplicity. (This corresponds to choosing one of $j_k$ cycles.) We also record the new maximum edge cycle length if it has changed. The latter two operations, i.e. compensating for overcount and recording the new maximum edge cycle length are common to all cases and will not be discussed further.

```
  // two vertices on the same cycle
  for(vlen1=2; vlen1<=vmxlen; vlen1++){
    inst1 = [[vmset objectAtIndex:vlen1-1] intValue];
    if(inst1){
      if(vlen1%2){
        elen = vlen1;
        instcount = vlen1*(vlen1-1)/2;
        instcount *= inst1;
```

```
            instcount /= elen;
            AddToMSet(emset, elen, instcount);
            if(elen>upper){
               upper = elen;
            }
         }
```

The next case corresponds to two vertices being located on a cycle of even length. Edges whose vertices are directly accross from one another need only move through half the cycle to return to the start position. This case is handled in the first half of the clause. The second half corresponds to vertex pairs that are not directly accross from one another and need to move through the entire cycle.

```
         else{
            elen = vlen1/2;
            instcount = vlen1/2;
            instcount *= inst1;

            instcount /= elen;
            AddToMSet(emset, elen, instcount);
            if(elen>upper){
               upper = elen;
            }

            elen = vlen1;
            instcount = vlen1*(vlen1-1)/2-vlen1/2;
            instcount *= inst1;

            instcount /= elen;
            AddToMSet(emset, elen, instcount);
            if(elen>upper){
               upper = elen;
            }

         }
      }
   }
```

The second case occurs when two vertices lie on different cycles that have the same length. We can choose these two cycles in $1/2j_k(j_k - 1)$ ways. The two vertices move in parallel along the two cycles and return to the start point only after having completed the whole vertex cycle.

```
   // two vertices on different cycles of the same length
   for(vlen1=1; vlen1<=vmxlen; vlen1++){
      inst1 = [[vmset objectAtIndex:vlen1-1] intValue];
      if(inst1>1){
         elen = vlen1;
         instcount = vlen1*vlen1;
         instcount *= inst1*(inst1-1)/2;

         instcount /= elen;
         AddToMSet(emset, elen, instcount);
         if(elen>upper){
            upper = elen;
         }
      }
   }
```

The last case occurs when the two vertices lie on different cycles of different lengths. We check all possible combinations. We must process all pairs $(j_{k_1}, j_{k_2})$ where $j_{k_1}$ and $j_{k_2}$ are not zero.

```
   // two vertices on different cycles of different lengths
   for(vlen1=1; vlen1<=vmxlen; vlen1++){
      for(vlen2=vlen1+1; vlen2<=vmxlen; vlen2++){
         inst1 = [[vmset objectAtIndex:vlen1-1] intValue];
         inst2 = [[vmset objectAtIndex:vlen2-1] intValue];

         if(inst1 && inst2){
```

The number of steps is the LCM of the lengths of the two cycles. E.g. for a cycle of length 2 and one of length 3, we visit 11, 22, 13, 21, 12, 23 and 11, a total of six steps. There are $k_1 k_2$ possible edges for each of $j_{k_1} j_{k_2}$ pairs of vertex cycles.

```
            elen = vlen1*vlen2/gcd(vlen1, vlen2);
            instcount = vlen1*vlen2;
            instcount *= inst1*inst2;

            instcount /= elen;
            AddToMSet(emset, elen, instcount);
            if(elen>upper){
               upper = elen;
            }
         }
      }
   }
```

The edge permutation shape is now ready. We remove any zeros that may remain at the upper end.

```
   if(upper<emxlen){
      [emset
         removeObjectsInRange:
            NSMakeRange(upper, emxlen-upper)];
   }

   return emset;
}
```

The remaining functions implement basic arithmetic for polynomials in one variable. We use algorithms that suffice for our needs. E.g. we do not implement polynomial multiplication by the FFT.

### 1.5.8 Implementation II: Polynomials

Polynomials are represented by arrays of `GMPInt` coefficients. Start with a very basic polynomial, namely the term $f(z^k)$, i.e. $1 + z^k$. We set all coefficients except for the constant term and $z^k$ to zero.

```
NSMutableArray *PolyK(int k)
{
   NSMutableArray *p
      = [NSMutableArray arrayWithCapacity:k+1];
   int l;
   GMPInt *zero = [GMPInt zeroObj], *one = [GMPInt oneObj];

   [p addObject:one];
   for(l=1; l<k; l++){
      [p addObject:zero];
   }
   [p addObject:one];
```

```
    return p;
}
```

We add polynomials by iterating over their coefficients and computing the sum of the coefficients of like powers.

```
NSMutableArray *PolySum(NSMutableArray *p1, NSMutableArray *p2)
{
    int c1 = [p1 count], c2 = [p2 count];
    int mx = (c1>c2 ? c1 : c2), pos;
    GMPInt *zero = [GMPInt zeroObj];

    NSMutableArray *sum =
        [NSMutableArray arrayWithCapacity:mx];

    for(pos=0; pos<mx; pos++){
```

Polynomials of different degrees are handled by checking whether the current power is present in the polynomial and using a zero value otherwise.

```
        GMPInt
            *n1 = (pos < c1 ? [p1 objectAtIndex:pos] : zero),
            *n2 = (pos < c2 ? [p2 objectAtIndex:pos] : zero),
            *res = [n1 add:n2];
        [sum addObject:res];
    }

    return sum;
}
```

The product of two polynomials is computed by iterating over all possible pairs of terms. The terms $az^n$ and $bz^m$ yield $abz^{m+n}$. We start by computing the degree of the product. The coefficient array must have one more slot than this value (for the constant term). We initialize the product to be empty.

```
NSMutableArray *PolyProd(NSMutableArray *f1, NSMutableArray *f2)
{
    int c1 = [f1 count], c2 = [f2 count];
    int cap = c1+c2-1, p, p1, p2;
    GMPInt *zero = [GMPInt zeroObj];

    NSMutableArray *prod =
        [NSMutableArray arrayWithCapacity:cap];

    for(p=0; p<cap; p++){
        [prod addObject:zero];
    }
```

We iterate over the terms of the two polynomials. We have a term for the product when both terms have non-zero coefficients.

```
    for(p1=0; p1<c1; p1++){
        for(p2=0; p2<c2; p2++){
            GMPInt
                *n1 = [f1 objectAtIndex:p1],
                *n2 = [f2 objectAtIndex:p2];
            if([n1 isZero]==NO && [n2 isZero]==NO){
```

If this is the case, then we compute the degree of the product and its coefficient, read the old value for the degree, increment it, and write it back into the coefficient array of the product. We return the result when we are done.

```
            int ex = p1+p2;
            GMPInt
               *prev = [prod objectAtIndex:ex],
               *p = [n1 mul:n2],
               *cur = [prev add:p];
            [prod replaceObjectAtIndex:ex
                withObject:cur];
         }
      }
   }

   return prod;
}
```

We need to be able to compute powers of polynomials $p$ because a term like $a_k^m$ in the cycle index requires us to compute $(1+z^k)^m$. This particular example can be computed with the binomial theorem, but we use a generic routine that uses $\log m$ multiplications. The base cases are first: one when the exponent is zero and $p$ when the exponent is one.

```
NSMutableArray *PolyPow(NSMutableArray *p, unsigned int ex)
{
   NSMutableArray *factor = p,
      *res =
      [NSMutableArray
         arrayWithObject:[GMPInt oneObj]];

   if(!ex){
      return res;
   }

   if(ex==1){
      return p;
   }
```

We now process the bits of the exponent in turn, starting with the least significant bit. This is a folklore algorithm, too. If the bit $b$ is set, then we include $p^{2^b}$ in the product. We obtain $p^{2^b}$ by squaring $p^{2^{b-1}}$.

```
   while(ex){
      if(ex%2){
         res = PolyProd(res, factor);
      }
      ex >>= 1;
      factor = PolyProd(factor, factor);
   }

   return res;
}
```

The next two routines in this section are very similar. We use them either to multiply the coefficients of a polynomial by some integer value, or to divide them by an integer value.

Multiplication is first. We iterate over the terms with a loop.

```
NSMutableArray *PolyScaleMul(NSMutableArray *p, GMPInt *f)
{
```

```
   int pos, mx = [p count];
   NSMutableArray *res =
      [NSMutableArray arrayWithCapacity:mx];
   GMPInt *zero = [GMPInt zeroObj];

   for(pos=0; pos<mx; pos++){
```

If we find a non-zero coefficient, then we multiply it by the factor $f$ and record the new value.

```
      GMPInt
         *prev = [p objectAtIndex:pos],
         *cur = zero;
      if([prev isZero]==NO){
         cur = [prev mul:f];
      }
      [res addObject:cur];
   }

   return res;
}
```

Division works the same way. We will use these two routines to scale terms of the cycle index by their coefficients and average the substituted cycle index at the very end.

```
NSMutableArray *PolyScaleDiv(NSMutableArray *p, GMPInt *f)
{
   int pos, mx = [p count];
   NSMutableArray *res =
      [NSMutableArray arrayWithCapacity:mx];
   GMPInt *zero = [GMPInt zeroObj];

   for(pos=0; pos<mx; pos++){
      GMPInt
         *prev = [p objectAtIndex:pos],
         *cur = zero;
      if([prev isZero]==NO){
         cur = [prev div:f];
      }
      [res addObject:cur];
   }

   return res;
}
```

At this point we have all the support that needs to be in place for us to be able to actually substitute $1 + z$ into a term from the cycle index. We must replace $a_k$ by $(1 + z^k)$ in the product $\prod_{k=1}^{p} a_k^{j_k}$. This is what the next routine does. It iterates over the $j_k$ and looks for nonzero values.

```
NSMutableArray *MSetToPoly(NSMutableArray *mset)
{
   NSMutableArray
      *res =
      [NSMutableArray
         arrayWithObject:[GMPInt oneObj]];

   int pos, mx = [mset count];

   for(pos=0; pos<mx; pos++){
```

```
      int e = [[mset objectAtIndex:pos] intValue];
      if(e){
```

If it finds such a value, then it includes the term $(1+z^k)^{j_k}$ in the product. The last step is to return the result. E.g. this will transform $a_1\,a_2^3$ into $z^7+z^6+3z^5+3z^4+3z^3+3z^2+z+1$.

```
         res = PolyProd(res, PolyPow(PolyK(pos+1), e));
      }
   }

   return res;
}
```

We promised extensive debug facilities and hence we need to be able to convert polynomials into strings that we can output during debugging. We do this in the next routine by computing the individual terms and joining them by interpolating plus signs. We use the LaTeX format, which is readable and ready for inclusion into LaTeX documents.

We iterate over the terms and skip empty ones, processing the others and storing the coefficient string in the variable `cfStr`.

```
NSString *PolyToString(NSMutableArray *p)
{
   int pos, mx = [p count];
   NSMutableArray *terms =
      [NSMutableArray arrayWithCapacity:mx];

   for(pos=0; pos<mx; pos++){
      GMPInt *coeff = [p objectAtIndex:pos];

      if([coeff isZero]==YES){
         continue;
      }

      NSString
         *cfstr = [coeff stringValue], *term;
```

The constant term is identical to its coefficient.

```
      if(!pos){
         term = cfstr;
      }
```

The term for $z$ is either $z$ when the coefficient $q$ is one, and $qz$ otherwise.

```
      else if(pos==1){
         if([coeff isOne]==YES){
            term = @"z";
         }
         else{
            term =
               [NSString stringWithFormat:
                         @"%@z", cfstr];
         }
      }
```

The term for $qz^k$ where $k>1$ is $z^k$ when $q$ is one and $qz^k$ otherwise.

```
      else{
         if([coeff isOne]==YES){
```

```
            term =
               [NSString stringWithFormat:
                         @"z^{%d}", pos];
         }
         else{
            term =
               [NSString stringWithFormat:
                         @"%@ z^{%d}", cfstr, pos];
         }
      }

      [terms addObject:term];
   }
```

The routine returns the terms, joined by a plus sign.

```
   return [terms componentsJoinedByString:@" + "];
}
```

### 1.5.9 Implementation III: Pólya's theorem

Counting graphs is easy with the set of routines that we have presented and we can concentrate on the high-level aspects of the algorithm. Our program needs a bit of an interface; it will process the command line and accept two arguments, the first of which is an optional switch `-d`, which turns on debugging, and the second or third gives $n$, the number of vertices of the graphs.

We need an autorelease pool as explained earlier. We also need the command line arguments if we want to process them.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   NSProcessInfo *procInfo = [NSProcessInfo processInfo];
   NSArray *args = [procInfo arguments];
   int ac = [args count];

   NSString *nstr;
```

The first thing we do before any other processing is to clear the debug flag and declare an array of classes whose memory allocation we wish to debug. In the case of mutable arrays and strings we obtain the class from an instance rather than the class itself. This is because these two classes produce instances whose class is not the same as the class that was asked to instantiate itself. We also initialize an array of allocation counters that allow us to test when allocation has doubled and empty the autorelease pool if this is the case. The variable `cl` is used to iterate over the array of classes.

```
   BOOL debug = NO;

   Class todebug[] = {
      [GMPInt class],
      [[NSMutableArray array] class],
      [[NSString stringWithFormat:@"instance %d", 0] class],
      nil
      }, *current;
   int allocs[] = {
      0, 0, 0
   };
   GSDebugAllocationActive(YES);
```

```
BOOL gc; id cl; int cind;
```

We still have some basic command line processing to do. We require two or three arguments; everything else is an error. (The arguments include the program name.)

```
if(ac!=2 && ac!=3){
   [NSException raise:NSInvalidArgumentException
            format:@"usage: %@ [-d] <n>",
            [procInfo processName]];
}
```

If we have three arguments, then the second one must be the debug flag. We raise an exception if it isn't.

```
if(ac==3 &&
  [[args objectAtIndex:1] isEqualToString:@"-d"]==NO){
   [NSException raise:NSInvalidArgumentException
            format:@"'-d' expected, got %@",
            [args objectAtIndex:1]];
}
```

We turn debugging on if we did get a debug flag and extract the string for $n$ from the third position.

```
else if(ac==3){
   nstr = [args objectAtIndex:2];
   debug = YES;
}
```

The string for $n$ is in the second position if there was no debug flag.

```
else{
   nstr = [args objectAtIndex:1];
}
```

We instantiate a scanner so that we can extract the value from the argument string for $n$. We raise an exception if the scanner could not read an integer or if it wasn't positive. Otherwise we initialize `t`, which is the number of edges.

```
NSScanner *scn = [NSScanner scannerWithString:nstr];
int n = -1;
if([scn scanInt:&n]==NO || n<1){
   [NSException raise:NSInvalidArgumentException
            format:@"positive vertex count, please; "
            @"got %d", n];
}
int t = n*(n-1)/2, s;
```

We can now prepare for the computation. We compute $n!$ and store the value in $f$, retaining $f$. We need $f$ as the numerator of the coefficient of a product term in the cycle index and to check that the coefficients of those terms really do add up to $n!$ This is the purpose of the checksum.

```
GMPInt *f = [GMPInt factorialUI:n],
   *vertexCheckSum = [GMPInt zeroObj];
[f retain]; [vertexCheckSum retain];
```

We will be iterating over partitions, turning them into vertex permutations first and then edge permutations, finally substituting them with $1 + z$. The variable `psum` holds the sum of the terms that we have processed so far. It is a polynomial and starts out being zero.

```
NSMutableArray *psum =
   [NSMutableArray
```

```
        arrayWithObject:
           [GMPInt zeroObj]];
   [psum retain];
```

We are ready for the first step, i.e. computing the partitions. We store and retain the result so that it will not be released. We must also retain the enumerator that we use to iterate over the partitions. We may now start the iteration itself.

```
   NSMutableArray *part, *parts = Partitions(n);
   [parts retain];

   NSEnumerator *en = [parts objectEnumerator];
   [en retain];

   while((part = [en nextObject])!=nil){
```

First we convert the partition to a vertex permutation shape, then we convert that shape in turn into an edge permutation shape. We compute the denominator of the coefficient. The numerator is $n!$. This yields the coefficient.

```
      NSMutableArray
         *vmset = PartitionToMSet(part),
         *emset = VertexMSetToEdgeMSet(vmset);

      GMPInt *denom = MSetToCoeffDenom(vmset),
         *coeff = [f div:denom];
```

We add the coefficient to the checksum and do the substitution of $1 + z^k$ for $a_k$ in the edge permutation multiset.

```
      [vertexCheckSum autorelease];
      vertexCheckSum = [vertexCheckSum add:coeff];
      [vertexCheckSum retain];

      NSMutableArray *poly = MSetToPoly(emset);
```

We temporarily leave the computation in order to output debugging information if the user did activate the debug feature. We convert the current cycle multisets for the vertex and the edge permutation to strings and output them together with the coefficient and the polynomial we obtained.

```
      if(debug==YES){
         NSString
            *vprod = MSetToProduct(vmset),
            *eprod = MSetToProduct(emset);

         fprintf(stderr, "%s␣[␣%s␣]␣[␣%s␣]␣%s\n",
               [[coeff description] cString],
               [vprod cString],
               [eprod cString],
               [PolyToString(poly) cString]);
      }
```

Now return to the computation. We release the old sum of the substituted cycle index terms, compute the new sum, and retain it.

```
      [psum autorelease];
      psum = PolySum(psum, PolyScaleMul(poly, coeff));
      [psum retain];
```

Two important consistency checks are next. The cycle lengths for the vertex permutation must add up to $n$, anything else is an error.

```
    if((s=SumIt(vmset))!=n){
       [NSException
          raise:NSInternalInconsistencyException
          format:@"vertices don't add up: %d, %d",
          n, s];
    }
```

Similarly, the cycle lengths for the edge permutation must add up to $t = 1/2n(n-1)$.

```
    if((s=SumIt(emset))!=t){
       [NSException
          raise:NSInternalInconsistencyException
          format: @"edges don't add up: %d, %d",
          t, s];
    }
```

We leave the computation for the second time, this time in order to release objects that are no longer used. We start by setting the release flag to off and compare the number of allocated objects of the three classes that we track to the number of allocations that we have recorded for them. If this value has more than doubled, then it is time to release the pool.

```
    gc = NO;
    current=todebug;
    while(*current!=nil){
       cind = current-todebug;
       id cl=*current++;

       if(2*allocs[cind]<GSDebugAllocationCount(cl)){
          gc = YES;
          break;
       }
    }
```

We release the pool if necessary and record the new values for the number of allocated objects from each class. This ends the body of the loop and hence the actual computation.

```
    if(gc==YES){
       if(debug==YES){
          fputs("GC ...\n", stderr);
       }
       [pool release];
       pool = [NSAutoreleasePool new];

       current=todebug;
       while(*current!=nil){
          cind = current-todebug;
          id cl=*current++;

          allocs[cind] = GSDebugAllocationCount(cl);
       }
    }
}
```

We subtract the actual total for the coefficients from $n!$ and record all three values (n!, checksum, difference) in a string.

```
   GMPInt *delta = [f sub:vertexCheckSum];
   NSString *csinfo =
      [NSString stringWithFormat:@"expected %@, "
              @"got %@, difference %@",
              [f stringValue],
              vertexCheckSum, delta];
```

We raise an exception if the checksum and $n!$ do not match.

```
   if([delta isZero]==NO){
         [NSException
            raise:NSInternalInconsistencyException
            format: @"checksum: %@", csinfo];
   }
```

We print some statistics if the user requested them, namely the checksum summary and the allocation counts of all classes that we are tracking.

```
   if(debug==YES){
      fprintf(stderr, "\nchecksum: %s\n\n", [csinfo cString]);

      current=todebug;
      while(*current!=nil){
         cl=*current++;
         fprintf(stderr, "%s count %u total %u peak %u\n",
               [NSStringFromClass(cl) cString],
               GSDebugAllocationCount(cl),
               GSDebugAllocationTotal(cl),
               GSDebugAllocationPeak(cl));
      }
   }
```

We are done. It remains to average the cycle index by dividing the sum of the polynomial terms by $n!$ and print the result. This is the generating function that we are looking for. The term $qz^k$ where $1 \leq k \leq \frac{1}{2}n(n-1)$ tells us that there are $q$ nonisomorphic graphs with $k$ edges on $n$ vertices.

```
   printf("%s\n",
        [PolyToString(PolyScaleDiv(psum, f)) cString]);

   [pool release];
   exit(0);
}
```

### 1.5.10 Results

These are the first few generating functions $g_n$ for small $n$.

$$\begin{aligned}
g_2 &= 1 + z \\
g_3 &= 1 + z + z^2 + z^3 \\
g_4 &= 1 + z + 2z^2 + 3z^3 + 2z^4 + z^5 + z^6 \\
g_5 &= 1 + z + 2z^2 + 4z^3 + 6z^4 + 6z^5 + 6z^6 \\
&+ 4z^7 + 2z^8 + z^9 + z^{10} \\
g_6 &= 1 + z + 2z^2 + 5z^3 + 9z^4 + 15z^5 + 21z^6 \\
&+ 24z^7 + 24z^8 + 21z^9 + 15z^{10} + 9z^{11}
\end{aligned}$$

$$
\begin{aligned}
&+ \quad 5z^{12} + 2z^{13} + z^{14} + z^{15} \\
g_7 &= 1 + z + 2z^2 + 5z^3 + 10z^4 + 21z^5 \\
&+ \quad 41z^6 + 65z^7 + 97z^8 + 131z^9 + 148z^{10} \\
&+ \quad 148z^{11} + 131z^{12} + 97z^{13} + 65z^{14} + 41z^{15} \\
&+ \quad 21z^{16} + 10z^{17} + 5z^{18} + 2z^{19} + z^{20} + z^{21} \\
g_8 &= 1 + z + 2z^2 + 5z^3 + 11z^4 + 24z^5 \\
&+ \quad 56z^6 + 115z^7 + 221z^8 + 402z^9 + 663z^{10} \\
&+ \quad 980z^{11} + 1312z^{12} + 1557z^{13} + 1646z^{14} \\
&+ \quad 1557z^{15} + 1312z^{16} + 980z^{17} + 663z^{18} \\
&+ \quad 402z^{19} + 221z^{20} + 115z^{21} + 56z^{22} \\
&+ \quad 24z^{23} + 11z^{24} + 5z^{25} + 2z^{26} + z^{27} + z^{28}
\end{aligned}
$$

And here is an excerpt from the result for $n = 32$.

$$
\begin{aligned}
&\ldots \\
&+ \quad 34761657216148743448344973243138057890667300337466944z^{72} \\
&+ \quad 20046162645933656584598068158804382026728876017758784 0z^{73} \\
&+ \quad 11386484793983475548892545192739515466786802538997862 57z^{74} \\
&+ \quad 63710206329224191336379147980150587083508964164892052 95z^{75} \\
&+ \quad 35117687386619298252758128999284099659276881466816638923z^{76} \\
&+ \quad 19071274566057765341139957306708805630849705768167199832 3z^{77} \\
&+ \quad 10204975809801846356519312551652387949972447666081145753 02z^{78} \\
&+ \quad 53810366252837429586773778530296090552605790188163864558 44z^{79} \\
&+ \quad 27963157472855646088638121146117229104449204522412651394430z^{80} \\
&\ldots
\end{aligned}
$$

## 1.6 Backtracking: The n-queens problem

*by Marko Riedel*

### 1.6.1 Idea

The n-queens problem consists in placing n non-attacking queens on an n-by-n chess board. A queen can attack another queen vertically, horizontally, or diagonally. E.g. placing a queen on a central square of the board blocks the row and column where it is placed, as well as the two diagonals (rising and falling) at whose intersection the queen was placed.

The algorithm to solve this problem uses backtracking, but we will unroll the recursion. The basic idea is to place queens column by column, starting at the left. New queens must not be attacked by the ones to the left that have already been placed on the board. We place another queen in the next column if a consistent position is found. All rows in the current column are checked. We have found a solution if we placed a queen in the rightmost column. A solution to the five-queens problem is shown in Figure 4.

Recursion is the obvious choice to implement this algorithm. We have an additional requirement, however. The user should be able to run the search step by step as well as automatically. E.g. she may want to halt the search at the middle column and observe how subsequent queens are placed. We would have to somehow stop and restart a sequence of nested calls if recursion were used. We record the state of the search instead. First, we record where queens have so far been placed on the board. Second, we record which row of the rightmost column we are currently checking.

The GUI is simple. We draw the board and the queens that have so far been placed. Placed queens are drawn in black. The board is drawn in blue during backtracking and in green if a solution was found. The

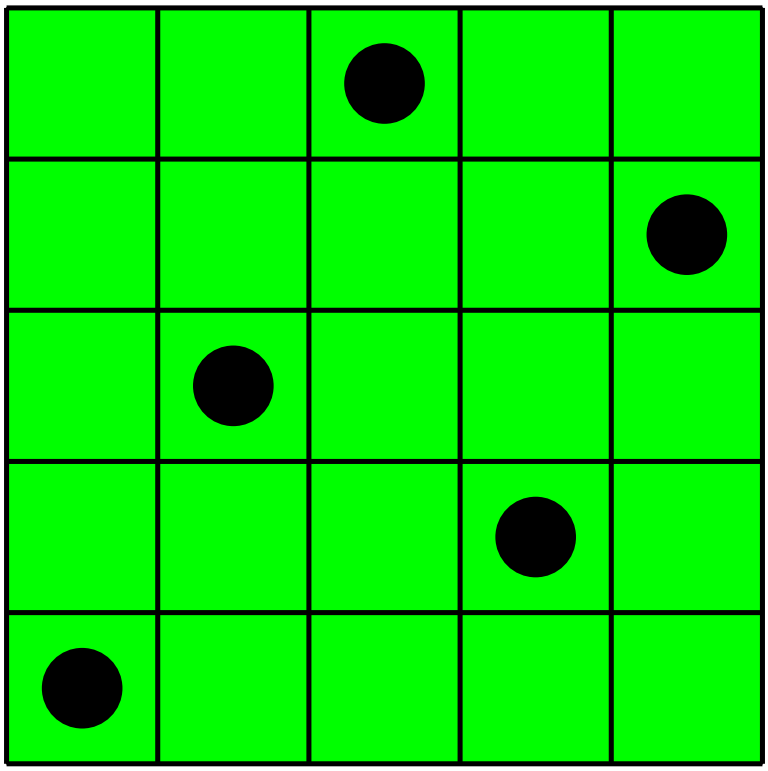

Figure 4: Solution to the five-queens problem.

queen that we are currently trying to place is drawn in red. The application's menu lets the user single-step through the search or choose to run it automatically. The algorithm returns to the initial position after all positions have been searched.

The application's structure is straightforward. There are two classes: a controller and a subclass of `NSView`. The controller is the application's delegate. It parses command-line arguments, sets up and responds to the main menu and controls the timer that is used during automated searches. The view draws the board and knows how to move from one state of the search to the next.

### 1.6.2 Implementation

We start by including the usual headers and define a constant that determines the width in pixels of a single cell of the board. The class `QView:NSView` represents and draws the board. Its instance variables store the size of the board, the number of solutions found during the current run and the state of the search. The latter is determined by the array `board`, whose indices correspond to columns and values to rows. It records where queens have been placed. The value of `candidate` records where in the rightmost empty column we are trying to place the next queen.

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

#define CELLSIZE 60

@interface QView : NSView
{
    int d;
    NSMutableArray *board;
    int candidate;
    int solCount;
```

```
}
```

There are only a few methods in the implementation of `QView`. The initializer creates a view of the right size for the given dimension (width/height). The method `drawRect:` is where the view draws itself. The most important method is `next`, which advances from one state of the search to the next. There is a method that returns a boolean that indicates whether the current state of the board represents a solution.

```
- initWithDimension:(int)dim;
- (void)drawRect:(NSRect)aRect;


- next;

- (BOOL)isSolution;

@end
```

The initializer is straighforward. It invokes the initializer of its superclass `NSView` with a rectangle of the appropriate dimensions. It creates the mutable array that holds the board. It is empty because initially there is no queen on the board. We start at column zero, with the bottom position being the first of `d` positions that must be checked, hence the value of `candidate` is zero. The counter for the number of solutions that have been found is set to zero.

```
@implementation QView

- initWithDimension:(int)dim
{
   d = dim;
   [super initWithFrame:NSMakeRect(0, 0, d*CELLSIZE, d*CELLSIZE)];

   board = [NSMutableArray arrayWithCapacity:d];
   [board retain];

   candidate = 0;

   solCount = 0;

   return self;
}
```

The method `drawRect` is `NSView`'s designated draw method where the view draws itself. The background of the board is green if the current state of the board represents a solution and blue otherwise. The variable `mx` holds the width and height of the view in pixels.

```
- (void)drawRect:(NSRect)aRect
{
   int mx = d*CELLSIZE;

   [([self isSolution] ?
     [NSColor greenColor] : [NSColor blueColor]) set];
   PSrectfill(0, 0, mx, mx);
```

The next step is to draw the grid, which consists of $2n+2$ lines, $n+1$ horizontal ones and $n+1$ vertical ones. The loop draws the lines in pairs, moving in parallel from bottom to top and from left to right.

```
   [[NSColor blackColor] set];
   PSsetlinewidth(2.0);
   int l;
   for(l=0; l<=mx; l+=CELLSIZE){
```

```
      PSmoveto(0, l);
      PSlineto(mx, l);
      PSmoveto(l, 0);
      PSlineto(l, mx);
   }
   PSstroke();
```

We now draw the queens using the data stored in the array `board`. A queen is represented by a black filled circle whose radius is a quarter of the cell size. The queen is centered in the respective cell.

```
   int col, placed = [board count];
   for(col=0; col<placed; col++){
      PSarc(col*CELLSIZE+
           CELLSIZE/2,
           [[board objectAtIndex:col] intValue]*CELLSIZE+
           CELLSIZE/2,
           CELLSIZE/4, 0, 360);
      PSfill();
   }
```

It remains to draw the queen that is currently being checked. It is farthest to the right and hence its column index is the number of queens already placed. The value of `candidate` determines the row.

```
   if(placed<d){
      [[NSColor redColor] set];
      PSarc(placed*CELLSIZE+
           CELLSIZE/2,
           candidate*CELLSIZE+
           CELLSIZE/2,
           CELLSIZE/4, 0, 360);
      PSfill();
   }
}
```

We move on to the most important part of this recipe, i.e. the method that takes the search from one state to the next. We first extract the number of queens that have been placed on the board. The current candidate resides in the column to the right of the rightmost queen, where `candidate` determines in what row the candidate is placed. The latter value increases during the search. It starts at the bottom row and is incremented until the top row is reached, one increment per state change.

```
- next
{
   int placed = [board count];
```

The candidate is admissible only if it is consistent with the queens that are already on the board, i.e. if it is not attacked by/does not attack any of these queens. We check each of them in turn. If the candidate is in the same row or column or diagonal then the position is not consistent. Rows and columns are easy to check. Use the following observation for diagonals. A falling diagonal has the equation $y = -x + p$. Two queens that lie on the same falling diagonal have the same value for $y+x$. A rising diagonal has the equation $y = x + p$. Two queens that lie on the same rising diagonal have the same value for $y - x$. A candidate is consistent if it passes the check for all queens that are already on the board.

```
   BOOL consistent = YES;
   int col;
   for(col=0; col<placed; col++){
      int row = [[board objectAtIndex:col] intValue];
      if(row==candidate ||
        col==placed ||
        row+col == candidate+placed ||
```

```
        row−col == candidate−placed){
         consistent = NO;
         break;
      }
   }
```

If the candidate's position is consistent, then we place a queen on the board and position the next candidate at the very bottom of the next column. The consistency check is also run if a solution has been found. It fails in this case because all available rows are blocked. Hence no new extra queen is ever added to the right of the right side of the board, and a solution leads to the else clause, where we increment the solution count, remove the rightmost successfully placed queen and make the cell just above it the new candidate we must check. We move the candidate up one cell otherwise, i.e. when the position is not consistent.

```
   if(consistent==YES){
      [board addObject:[NSNumber numberWithInt:candidate]];
      candidate = 0;
   }
   else{
      if(placed==d){
         candidate = [[board lastObject] intValue]+1;
         [board removeLastObject];
         solCount++;
      }
      else{
         candidate++;
      }
```

We backtrack to the rightmost column that has not been checked completely. A row value of $d$ indicates that a column has been checked. We move backwards from the right to the left, eliminating columns that we have checked. The new candidate position in a given column is one cell above the previous position, hence the increment.

```
      while(candidate==d && [board count]){
         candidate = [[board lastObject] intValue]+1;
         [board removeLastObject];
      }
```

There remains some housekeeping to do. We can detect the transition from the last state to the first by checking whether the previous state was the top position of the first column. If this is the case, then we can output the number of solutions found and reset the solution counter to zero.

```
      if(![board count] && candidate==d){
         candidate = 0;
         NSLog(@"found␣%d␣solutions␣to␣the␣%d−queens␣problem",
              solCount, d);
         solCount = 0;
      }
   }
```

The method triggers a redisplay at the end since the state of the board has certainly changed.

```
   [self display];
   return self;
}
```

Sometimes the controller needs to know whether the current state of the board represents a solution or not. This is the purpose of `isSolution`. We have a solution if the number of queens on the board equals the number of rows/columns of the board.

```
- (BOOL)isSolution
{
    return([board count]==d);
}

@end
```

We are now ready to move on to the controller. Recall that the user may step through a search or run it automatically. The latter case requires a timer. We define two constants that determine how much time elapses between firings of the timer. It fires quickly (after 0.15 seconds) if the current position does not represent a solution, and waits a full five seconds otherwise.

```
#define FAIL_PAUSE 0.15
#define SOL_PAUSE 5
```

The controller obviously needs to keep track of the `QView`. It also stores whether the search is currently running automatically or not. There is an invocation for use with the timer (we use the same invocation again and again). The controller also stores the timer.

```
@interface Controller : NSObject
{
    QView *qv;

    BOOL running;

    NSInvocation *inv;
    NSTimer *tm;
}
```

The controller is the application's delegate and `applicationDidFinishLaunching:` is its most important method, where it initializes the menu, creates the window with the board and prepares for use of the timer. The three methods `runSearch:`, `haltSearch:` and `next:` are invoked through clicks on the application's main menu. They run the search, halt it or advance it by a single step, respectively. The method `validateMenuItem:` enables and disables the corresponding entries on the main menu according to the run state of the application. The method `tick` is invoked when the timer fires.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif;

- runSearch:(id)sender;
- haltSearch:(id)sender;
- next:(id)sender;

- (BOOL)validateMenuItem:(NSMenuItem*)anItem;

- tick;

@end
```

The method `validateMenuItem:` recognizes menu items by tags. This is recommended for large applications where internationalization plays a role; i.e. we do not want to depend on the title of an item, as it may change. (Thanks Fabien Vallon for the hint.)

```
typedef enum {
    MENU_START,
    MENU_STOP,
    MENU_NEXT,
    MENU_QUIT
} MENU_TAG;
```

The first thing `applicationDidFinishLaunching:` does is to query the command line arguments. The second argument contains the size of the board.

```
@implementation Controller

- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   NSProcessInfo *procInfo = [NSProcessInfo processInfo];
   NSArray *args = [procInfo arguments];
```

The next step is to build the application's menu and set the tags of each item. There are three entries to control the search: run, halt and next. The fourth entry of the menu quits the application. We assign and display the menu.

```
   NSMenu *menu = [NSMenu new];

   [[menu addItemWithTitle: @"Run"
        action:@selector(runSearch:)
        keyEquivalent:@""] setTag:MENU_START];
   [[menu addItemWithTitle: @"Halt"
        action:@selector(haltSearch:)
        keyEquivalent:@""] setTag:MENU_STOP];
   [[menu addItemWithTitle: @"Next"
        action:@selector(next:)
        keyEquivalent:@""] setTag:MENU_NEXT];
   [[menu addItemWithTitle: @"Quit"
        action:@selector(terminate:)
        keyEquivalent:@"q"] setTag:MENU_QUIT];

   [NSApp setMainMenu:menu];

   [menu display];
```

We require exactly one argument, namely the size of the board.

```
   if([args count]!=2){
      [NSException raise:NSInvalidArgumentException
                  format:@"usage: %@ <n>",
                  [procInfo processName]];
   }
```

The command line argument is a string. We require a scanner to retrieve the integer value from the string. We throw an exception if the scanner fails or the board is to small (min. is a two by two).

```
   NSScanner *scn =
      [NSScanner scannerWithString:[args objectAtIndex:1]];
   int dim = -1;
   if([scn scanInt:&dim]==NO || dim<2){
      [NSException raise:NSInvalidArgumentException
                  format:@"integer board size >= 2, please; "
                  @"got %d", dim];
   }
```

The intial state is step mode, i.e. we are not running.

```
   running = NO;
```

We are ready to instantiate the `QView` that will perform the search and display the board.

```
   qv = [[QView alloc] initWithDimension:dim];
```

The next step is to build the window. The size of the content rectangle is equal to the size of the view, since it will be the content view of the window. We allocate and initialize the window, which is titled, but not resizable. The title of the window indicates what size of queens problem we are solving.

```
   NSWindow *qWin;
   NSRect winRect;

   winRect.origin = NSMakePoint(0, 0);
   winRect.size = [qv bounds].size;

   qWin = [[NSWindow alloc]
            initWithContentRect:winRect
            styleMask:NSTitledWindowMask
            backing:NSBackingStoreBuffered
            defer:NO];
   [qWin
      setTitle:
         [NSString stringWithFormat:@"%d_queens_problem", dim]];
```

We place the view in the window, center it and make it appear on screen.

```
   [qWin setContentView:qv];

   [qWin center];
   [qWin makeKeyAndOrderFront:nil];
```

The last step is to create the one invocation that will be used with all timers. It captures the target object (the controller) and the selector (`tick`). It is retained because it must persist throughout the lifetime of the application.

```
   inv = [NSInvocation
           invocationWithMethodSignature:
              [self methodSignatureForSelector:
                       @selector(tick)]];
   [inv setSelector:@selector(tick)];
   [inv setTarget:self];
   [inv retain];
}
```

The method `runSearch:` is invoked via the main menu. It sets the internal state to "running" and schedules a timer that fires at `FAIL_PAUSE` intervals.

```
- runSearch:(id)sender
{
   running = YES;
   tm = [NSTimer scheduledTimerWithTimeInterval:FAIL_PAUSE
              invocation:inv
              repeats:YES];

   return self;
}
```

The method `haltSearch:` resets the internal state to "halted" and invalidates the timer, which stops firing and is autoreleased.

```
- haltSearch:(id)sender
{
   running = NO;
   [tm invalidate];
```

```
    return self;
}
```

The method `next:` is invoked from the menu and advances the search one step.

```
- next:(id)sender
{
   [qv next];
   return self;
}
```

The method `validateMenuItem:` is invoked by the application object. It checks for three cases when a menu item should be disabled: the "stop" item when we are not running, the "start" item when we are running and the "next" item when we are running.

```
- (BOOL)validateMenuItem:(NSMenuItem*)anItem
{
   MENU_TAG tag = [anItem tag];

   if((tag==MENU_STOP && !running) ||
     (tag==MENU_START && running) ||
     (tag==MENU_NEXT && running)){
      return NO;
   }

   return YES;
}
```

The last method is called `tick` and it is invoked by the timer. It advances the search by a single step and checks whether a solution has been found, in which case the timer is set to fire after `SOL_PAUSE` seconds, so that the user has time to inspect the structure of the solution.

```
- tick
{
   [qv next];
   if([qv isSolution]){
      [tm setFireDate:
            [NSDate dateWithTimeIntervalSinceNow:SOL_PAUSE]];
   }

   return self;
}

@end
```

The function `main` concludes the recipe. It sets up an autorelease pool, creates the application object and makes the controller be its delegate. It then runs the application. The autorelease pool is released on exit.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   NSApplication *app;

   app = [NSApplication sharedApplication];
   [app setDelegate:[Controller new]];
   [app run];
```

```
    [pool release];
    exit(0);
}
```

# 2 Processing mouse events

## 2.1 Selecting a part of a static image

*by Marko Riedel*

### 2.1.1 Idea

This example is taken from a fractal viewer, where the fractal is stored in an image and the user wishes to select a circular region of the image for the purpose of zooming. It is not difficult to alter this code to select, say, a rectangular region.

The structure of the routine is simple. We override `NSView`'s `mouseDown:` method and process mouse-dragged events until we get a mouse-up event. We update the selection every time there is a mouse-dragged event.

### 2.1.2 Implementation

We must store two points, namely the start point, and the current point. The start point tells us where the center of the selection lies, and the larger of the two distances (horizontal, vertical) from the current point to the start point defines the radius of the circle being selected.

```
#define PAD 5

- (void)mouseDown:(NSEvent *)theEvent
{
    NSPoint startp, curp;
    NSEvent *curEvent = theEvent;

    float dx, dy, prevr=0, r;
    NSRect update;

    double deltare, deltaim,
        newminre, newmaxre, newminim, newmaxim;

    startp = [curEvent locationInWindow];
    startp = [self convertPoint:startp fromView: nil];
```

Both the start and the current point are supposed to be in view coordinates. Hence we must convert them from screen coordinates. We set the previous radius to zero, prepare the start point and are now ready to enter the loop.

```
    do {
        curp = [curEvent locationInWindow];
        curp = [self convertPoint:curp fromView: nil];

        dx = curp.x-startp.x;
        dy = curp.y-startp.y;

        if(dx<0){
            dx = -dx;
        }
        if(dy<0){
```

```
        dy = -dy;
    }

    r = (dx<dy ? dx : dy);
```

The first step is to convert the location of the drag event into the view's coordinates. We compute the radius next. We must restore the pixels that we altered when we drew the circle in the previous iteration of the loop, and draw the circle for the current radius. So we compute a square whose center lies at the start point and which includes the circle that was drawn. We copy the data from the image over this rectangle, thereby restoring the pixels that we altered.

```
    [self lockFocus];

    if(prevr){
        update=NSMakeRect(startp.x-prevr-PAD, startp.y-prevr-PAD,
                    2*(prevr+PAD), 2*(prevr+PAD));
        update=NSIntersectionRect(update, [self frame]);

        [image compositeToPoint:update.origin
             fromRect:update
             operation:NSCompositeCopy];
    }
```

The next step is to draw a circle centered at the start point and having the new radius. This code actually draws two circles, one black, one white and having a slightly larger radius. This assures that the user sees the circle even if it overlaps with a monochrome (white, black) portion of the image.

```
    PSsetgray(0.0);
    PSarc(startp.x, startp.y, r, 0, 360);
    PSstroke();
    PSsetgray(1.0);
    PSarc(startp.x, startp.y, r+1, 0, 360);
    PSstroke();

    [self unlockFocus];
    [[self window] flushWindow];

    prevr = r;
```

The last step is to flush the data and remember the new radius so that the underlying area can be restored. We process events until we get a mouse-up.

```
    curEvent =
        [[self window]
            nextEventMatchingMask:
                NSLeftMouseUpMask | NSLeftMouseDraggedMask];
} while([curEvent type] != NSLeftMouseUp);


deltare = maxre-minre;
deltaim = maxim-minim;
newminre = deltare*(double)(startp.x-r)/(double)res+minre;
newmaxre = deltare*(double)(startp.x+r)/(double)res+minre;
newminim = deltaim*(double)(startp.y-r)/(double)res+minim;
newmaxim = deltaim*(double)(startp.y+r)/(double)res+minim;

minre=newminre;
maxre=newmaxre;
```

```
    minim=newminim;
    maxim=newmaxim;

    [self update];
    [self setNeedsDisplay:YES];
}
```

We invoke `setNeedsDisplay:` so that the selected portion of the image will be displayed after it has been computed by `update`. We have to compute the coordinates for this to work. Assume that we are working with four values that describe a portion of the complex plane, i.e. real min and max and complex min and max; the variable `res` contains the width and height of the view. We convert the coordinates of the square centered at the start point into the coordinate system of the unit square, then convert into the coordinates of the square being displayed, which gives us the new square.

# 3 Working with tasks

## 3.1 User may interrupt read from task

*by Marko Riedel*

### 3.1.1 Idea

We want to collect output from a task, possibly after writing some data to its standard input. The caller of the method should have the option of a progress display that pops up after a certain time *and* allows the user to interrupt the task. The method returns an enumerator that iterates over the lines that were collected from the standard output of the task.

### 3.1.2 Implementation

We will need two pipes, one for writing to the standard input of the task and another for reading from its standard output. We also need a modal session for the progress display. We will use the `read` system call to perform non-blocking reads from `stdout`. There is a buffer where `read` will store data. All of these are declared below.

```
#define BUFSIZE 4096
#define INTERVAL  3

#define ERRMSG  "ERRGSMBrowser -- transfer interrupted"

- (NSEnumerator *)runCommand:(NSString *)cmd arguments:(NSArray *)args
                 input:(NSString *)inStr
              progress:(BOOL)prog
{
    NSPipe *inputPipe, *outputPipe;
    NSFileHandle *reader, *writer;
    NSMutableData *data = [NSMutableData dataWithCapacity:1024];
    NSTask *cmdTask = [NSTask new];

    int ticks = 0;

    int fd, len;
    char *buf[BUFSIZE];

    unsigned length;
    const char *bytes;

    NSApplication *app = [NSApplication sharedApplication];
```

```
NSModalSession progSession;
BOOL pflag = NO;
NSDate *progDate;

int mask = NSLeftMouseDownMask | NSLeftMouseUpMask |
   NSLeftMouseDraggedMask;
NSDate *evDate;
NSEvent *event;

NSLog(@"%@ %@", cmd, args);

outputPipe = [NSPipe pipe];
reader = [outputPipe fileHandleForReading];

fd = [reader fileDescriptor];
fcntl(fd, F_SETFL, O_NONBLOCK);

inputPipe = [NSPipe pipe];
writer = [inputPipe fileHandleForWriting];

[cmdTask setLaunchPath:cmd];
[cmdTask setArguments:args];

[cmdTask setStandardInput:inputPipe];
[cmdTask setStandardOutput:outputPipe];
[cmdTask setStandardError:outputPipe];
```

The first part of the routine allocates the two pipes and collects file handles for reading and writing, resp. Furthermore we activate non-blocking reads for the file for `stdout`. This key idea will allow us to animate the progress panel while we are reading data. We prepare for launch by setting the tasks `stdin` and `stdout` as well as the path to the executable and an array containing the arguments that are to be passed to the task. We are ready to launch the task. The first action is to write and close the task's `stdin`. We also compute a date some time in the future (three seconds after launch in this case.)

```
[cmdTask launch]; interrupted = NO;
progDate = [NSDate dateWithTimeIntervalSinceNow:INTERVAL];

if(inStr!=nil){
   [writer writeData:[inStr dataUsingEncoding:NSASCIIStringEncoding]];
}
[writer closeFile];

while(len = read(fd, buf, BUFSIZE)){
```

The return value of the `read` call tells us what occured: there are no data when it is less than zero, it is zero on EOF and gives the number of bytes if data were copied into the buffer. We simply store the data in an `NSData` object allocated for this purpose. We exit the loop at EOF.

The case when there are no data but we are not at EOF is special. We need to start a modal session should the caller have requested a progress display and we have been waiting longer than three seconds. Assume that the panel `progPanel` was created elsewhere in the application and contains a button that sets the flag `interrupted` to true. We start a modal session for the panel and set the cursor.

```
      if(len>0){
         [data appendBytes:buf length:len];
      }
      else{
         if(prog==YES && pflag==NO &&
```

```
          [progDate timeIntervalSinceNow]<0.0){
           [progPanel
              setTitle:[NSString stringWithFormat:@"Running␣%@", cmd]];

           progSession = [app beginModalSessionForWindow:progPanel];
           pflag = YES;
           [[NSCursor arrowCursor] push];
        }
```

We have to update the progress indicator should there be no data and the panel is already displayed. Here we assume that the `progInd` can display a value between zero and one hundred. We update the indicator and wait for 0.025 seconds whether the progress panel gets a mouse click event (mask defined above). We process the event if this is the case. This is where the user is able to click the stop button and set the flag `interrupted`. We terminate the task if we received an interrupt and set the data to contain an error message. The program sleeps for 0.025 seconds if there were no data, so that no busy-waiting occurs.

```
        else if(pflag==YES){
           ticks = (ticks+1)%200;
           [progInd
              setPercent:(float)(ticks<100 ? ticks : 199−ticks)];
           [progInd display];
           [progPanel flushWindow];

           evDate = [NSDate dateWithTimeIntervalSinceNow:0.025];
           event = [app nextEventMatchingMask:mask
                     untilDate:evDate
                     inMode:NSDefaultRunLoopMode
                     dequeue:YES];
           if([event window]==progPanel){
              [app sendEvent:event];
           }
           if(interrupted==YES){
              [cmdTask terminate];
              data = [NSData dataWithBytes:ERRMSG
                           length:strlen(ERRMSG)];
              break;
           }
        }
        [NSThread
           sleepUntilDate:
              [NSDate dateWithTimeIntervalSinceNow:0.025]];
     }
  }

  if(pflag==YES){
     [app endModalSession:progSession];
     [progPanel close];
     [NSCursor pop];
  }

  length = [data length];
  bytes = [data bytes];

  if(length && bytes[length−1]=='\n'){
     length−−;
  }
```

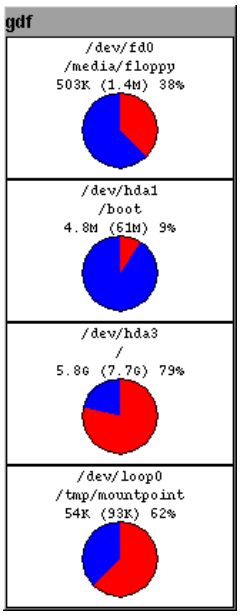


Figure 5: Graphical frontend to `df` displays file system disk usage.

```
    return
        [[[NSString stringWithCString:bytes length:length]
            componentsSeparatedByString:@"\n"]
           objectEnumerator];
}
```

The remainder of the code closes the progress panel if it was displayed and ends the modal session. The `return` statement splits the data from `stdout` into lines and returns an enumerator that can be used to iterate over those lines.

## 3.2 Frontend to `df`

*by Marko Riedel, from an idea by Martin Brecher*

### 3.2.1 Idea

The program `df` reports file system disk space usage. It displays these statistics in human-readable form when invoked with the `-h` option. Its output might look like this:

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda3             7.7G  5.8G  1.6G  79% /
/dev/hda1              61M  4.8M   53M   9% /boot
shmfs                 126M     0  126M   0% /dev/shm
/dev/loop0             93K   54K   34K  62% /tmp/mountpoint
/dev/fd0              1.4M  500K  840K  38% /media/floppy
```

The goal is to have a graphic frontend to `df` that displays the disk space usage for every disk in a graphical format like that shown in Fig.5. It should update from time to time in order to reflect dynamic changes in disk space usage.

### 3.2.2 Implementation

The program consists of two components: the first is the class `PercentageView`, which provides the functionality to display a couple of lines of data above a pie chart that represents the current percentage. We

will instantiate one `PercentageView` for every disk. The second component is a controller that acts as the application's delegate, runs `df` at one-second intervals and updates the percentages accordingly.

Start with some basic definitions that collect constants at the start of the program.

```
#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

// skeleton by Fred Kiefer

#define FONTSIZE    11
#define SEPARATOR    4

#define DFBIN @"/bin/df"

#define PWIDTH     160
#define PHEIGHT    100
```

These settings specify the font size to use, the distance from the pie to the margin, where `df` is to be found and the size of a single `PercentageView`.

A `PercentageView` needs to remember the percentage and the lines that make up the title, which we store in an `NSArray`.

```
@interface PercentageView : NSView
{
   NSArray *titles;
   int percentage;
}

- initWithTitles:(NSArray *)ts andFrame:(NSRect)frame;
- setTitles:(NSArray *)newts;
- setPercentage:(int)newpc;

- (void)drawRect:(NSRect)aRect;

@end
```

The interface declares the variables that hold the titles and the percentage. The first three methods permit to initialize a `PercentageView` with a given frame and a set of titles, as well as to set the percentage and the titles.

```
@implementation PercentageView

- initWithTitles:(NSArray *)ts andFrame:(NSRect)frame
{
   [super initWithFrame:frame];
   titles = ts;
   [titles retain];

   percentage = 0;

   return self;
}

- setTitles:(NSArray *)newts
{
   [titles release];

   titles = newts;
```

```
    [titles retain];
    [self setNeedsDisplay:YES];

    return self;
}

- setPercentage:(int)newpc
{
    if(newpc==percentage){
        return self;
    }

    percentage = newpc;
    [self setNeedsDisplay:YES];

    return self;
}
```

The core of this class lies is the method `drawRect:`, which actually draws the pie chart and the titles. We read the bounds rectangle so that we can use it to compute the coordinates of the components (text, chart). We get the font of the desired size and set the dictionary that contains the attributes of the title strings to contain just one entry, namely the font that we obtained. We need to iterate over the titles so we ask for the appropriate enumerator and use the variable `title` to hold a single line of the title section. The variable `height` is decremented by the height of a line every time we draw a line. It starts at the top of the view and moves downwards. The remaining variables pertain to the pie chart: they store the radius, what angle of the pie is occupied, and where the center of the chart lies. The variable `slice` is used to hold three different bezier paths that are used to draw the chart.

```
- (void)drawRect:(NSRect)aRect
{
    NSRect bounds = [self bounds];

    NSFont *font =
        [NSFont userFixedPitchFontOfSize:FONTSIZE];
    NSDictionary *attrs =
        [NSDictionary dictionaryWithObjectsAndKeys:
                      font, NSFontAttributeName, nil];

    NSEnumerator *titleEnum =
        [titles objectEnumerator];
    NSString *title;
    int height = 0;

    float radius;
    int angle = 360*percentage/100;
    NSPoint center;

    NSBezierPath *slice;
```

A `PercentageView` has a white background and a black boundary, which we draw first.

```
    [[NSColor whiteColor] set];
    [NSBezierPath fillRect:bounds];

    [[NSColor blackColor] set];
    [NSBezierPath strokeRect:bounds];
```

The next step is to iterate over the titles starting at the top of the view and draw each line in turn. We

compute the dimensions of each line and use the width to center the string and the height to decrement the variable `height`, i.e. the current position. Move the statement

```
height -= size.height;
```

to before the assignment of `loc.x` and `loc.y` if you have the corrected version of

```
drawAtPoint:withAttributes:,
```

which works like `PSmoveto()` and `PSshow()`.

```
while((title = [titleEnum nextObject])!=nil){
   NSSize size = [title sizeWithAttributes:attrs];
   NSPoint loc;

   loc.x = (bounds.size.width−size.width)/2;
   loc.y = height;
   [title drawAtPoint:loc withAttributes:attrs];

   height −= size.height;
}
```

The lower part of the view forms a box that will contain the chart. It is as wide as the view itself. Its height is given by the difference between the height of the view and the height of the titles that were drawn, i.e. the current value of the variable `height` gives the height of the box. We use the smaller of these two to determine the radius of the pie, so that it is sure to fit inside the view. The center of the pie lies in the middle of the x-axis and `radius+SEPARATOR` units below the last title that was drawn.

```
radius = (height < bounds.size.width ?
        height : bounds.size.width)/2
   − SEPARATOR;

center.x = bounds.size.width/2;
center.y = height−radius−SEPARATOR;
```

It remains to draw the pie chart. We draw three slices, starting with a red one that shows the part of the disk that is occupied, which is proportional to `angle`. The slice starts at "noon" and ends after it covers `angle` degrees going clockwise.

```
slice = [NSBezierPath bezierPath];
[slice moveToPoint:center];
[slice appendBezierPathWithArcWithCenter:center
     radius:radius
     startAngle:90 endAngle:90−angle
     clockwise:YES];
[slice closePath];

[[NSColor redColor] set];
[slice fill];
```

The second slice shows the free space in blue. The start and end angles are the same, except that we now draw in the opposite direction.

```
slice = [NSBezierPath bezierPath];
[slice moveToPoint:center];
[slice appendBezierPathWithArcWithCenter:center
     radius:radius
     startAngle:90 endAngle:90−angle
     clockwise:NO];
[slice closePath];
```

```
   [[NSColor blueColor] set];
   [slice fill];
```

The last step is to draw a black boundary around the chart. There are several ways to do this; we could use the code that we used to draw the red and the blue slice. Here is a different approach.

```
   slice =
      [NSBezierPath
         bezierPathWithOvalInRect:
            NSMakeRect(center.x−radius, center.y−radius,
                    2∗radius, 2∗radius)];

   [[NSColor blackColor] set];
   [slice stroke];
}

@end
```

We could have used PostScript operators to draw the view, but this recipe is intended to provide examples of drawing with the application kit.

The second main component is the controller. It stores the views in a dictionary, where the keys are strings, i.e. the devices being represented.

```
@interface Controller : NSObject
{
   NSMutableDictionary ∗devices;
}

− (NSMutableDictionary ∗)runDF;
− (NSArray ∗)infoToTitles:(NSArray ∗)devInf;
− update;
− (void)applicationDidFinishLaunching:(NSNotification ∗)notif;

@end
```

The method `runDF` is the heart of the controller. It runs `df` and returns a dictionary whose keys are the devices. Every key points to an array that contains the output from `df` for that device. The output entry in the dictionary is an array obtained by splitting the output line into fields, where fields are separated by spaces. The method `infoToTitles` turns such an array into an array of titles suitable for display in a `PercentageView`. The method `update` is invoked by a timer and invokes `runDF` in turn. It records and displays the changes. The last method is invoked when the application finishes launching. It assembles the window and displays an initial set of data.

The first thing `runDF` does is to create the dictionary that will hold the output lines for the devices. It declares an enumerator that iterates over the raw lines that we obtain from `df`. We will also need a character set containing spaces so that we can split lines into fields.

```
@implementation Controller

− (NSMutableDictionary ∗)runDF
{
   NSMutableDictionary ∗info =
      [NSMutableDictionary dictionaryWithCapacity:1];
   NSEnumerator ∗lineEnum;

   NSCharacterSet ∗space =
      [NSCharacterSet whitespaceAndNewlineCharacterSet];
```

The next step is to declare everything that we need to run `df` and obtain its output. The variables `length` and `bytes` will be set to the respective values that describe the data. We need a task object for `df` and a pipe from its standard output. The file handle `reader` enables us to read from the pipe. The data is read in chunks and appended to a mutable data object. The variable `line` holds a single line during processing of the output. We create a pipe and obtain the file handle for reading from that pipe.

```
unsigned length;
const char *bytes;

NSTask *dfTask = [NSTask new];
NSPipe *outputPipe;
NSFileHandle *reader;

NSData *chunk;
NSMutableData
   *data = [NSMutableData dataWithCapacity:1024];
NSString *line;

outputPipe = [NSPipe pipe];
reader = [outputPipe fileHandleForReading];
```

We must prepare the task before we can launch it. Therefore we set the actual binary that will be executed and add `-h` as an argument so that we get human-readable output from `df`. We set standard output and standard error to our pipe in order to read those data.

```
[dfTask setLaunchPath:DFBIN];
[dfTask setArguments:[NSArray arrayWithObject:@"-h"]];

[dfTask setStandardOutput:outputPipe];
[dfTask setStandardError:outputPipe];
```

Now we launch the task and read chunk after chunk until there are no more data. Actually, the output from `df` probably fits into a single chunk. We wait for the task to exit and close the file descriptor that is associated with the read end of the pipe so that it becomes available for re-use. Recall that we will invoke this method many times so as to present an up-to-date picture of file system usage.

```
[dfTask launch];
while((chunk = [reader availableData]) &&
     [chunk length]){
   [data appendData:chunk];
}
[dfTask waitUntilExit];

close([reader fileDescriptor]);
```

We turn the data into a string, split it into lines and obtain an enumerator of those lines.

```
length = [data length];
bytes = [data bytes];

lineEnum =
   [[[NSString stringWithCString:bytes length:length]
       componentsSeparatedByString:@"\n"]
      objectEnumerator];
```

The next step is to parse the data. We iterate over the array of lines and process those lines that begin with the string `/dev/`, i.e. refer to actual devices. We want to split the line into fields, so we create a scanner from the current line and prepare an empty array of fields. The variable `field` holds a single field, of which there are several per line.

```
   while((line = [lineEnum nextObject])!=nil){
      if([line hasPrefix:@"/dev/"]){
         NSScanner *scn =
            [NSScanner scannerWithString:line];
         NSMutableArray *fields =
            [NSMutableArray arrayWithCapacity:6];
         NSString *field;
```

We scan the fields in turn, until the scanner reaches the end. We skip leading whitespace, read a single field and store it in the array of fields for the current line. We record the fields in the dictionary when the entire line has been scanned.

```
         while([scn isAtEnd]==NO){
            [scn scanCharactersFromSet:space
                intoString:NULL];
            [scn scanUpToCharactersFromSet:space
                intoString:&field];
            [fields addObject:field];
         }

         [info setObject:fields
              forKey:[fields objectAtIndex:0]];
      }
   }
```

The routine checks if any data have successfully been read and raises an exception if there weren't any. It returns the dictionary whose keys are the devices and whose values hold the fields from the corresponding output line produced by `df`.

```
   if(![info count]){
      [NSException raise:NSGenericException
                format:@"no data from %@", DFBIN];
   }

   return info;
}
```

The method `infoToTitles` is very simple. It creates three title lines from the array of fields: the name of the device, the mountpoint and the current usage. The latter shows the current usage, the capacity and the percentage.

```
- (NSArray *)infoToTitles:(NSArray *)devInf
{
   NSString *devname, *mountpoint, *current;

   devname = [devInf objectAtIndex:0];
   mountpoint = [devInf objectAtIndex:5];

   current = [NSString stringWithFormat:@"%@ (%@) %@",
                  [devInf objectAtIndex:2],
                  [devInf objectAtIndex:1],
                  [devInf objectAtIndex:4]];

   return
      [NSArray arrayWithObjects:devname, mountpoint,
             current, nil];
}
```

The method `update` is invoked by the timer. It runs `df` to obtain the dictionary of lines split into fields and prepares an enumerator to iterate over the keys of the dictionary. The string `device` holds the current device.

```
- update
{
   NSMutableDictionary *info = [self runDF];
   NSEnumerator *devEnum;
   NSString *device;

   devEnum = [[info allKeys] objectEnumerator];
```

The actual iteration is next. For each device, do the following: extract the array of fields and look for the device in the dictionary that maps devices to views. If there is an entry for the device then there is a `PercentageView` for it that must be updated. We set the percentage from the fifth field and the set of titles as formatted by `infoToTitles`. This operation marks the view as needing redisplay. The method returns when all updates have been recorded.

```
   while((device = [devEnum nextObject])!=nil){
      NSArray *item = [info objectForKey:device];
      PercentageView *pview =
         [devices objectForKey:device];

      if(pview!=nil){
         [pview setPercentage:
                  [[item objectAtIndex:4] intValue]];
         [pview setTitles:[self infoToTitles:item]];
      }
   }

   return self;
}
```

The last method of the controller needs to prepare everything once the application has finished launching. It declares variables to hold the window that contains the percentage views, the main menu of the application, the dimensions of the window, and its content view. It invokes `runDF` to obtain the data that it needs to get started. It declares an enumerator in order to iterate over the devices and a string to hold the current device name. The variable `dcount` tracks the number of views that have been created. There is an invocation for use with the timer.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   NSWindow *dfWin;
   NSMenu *menu = [NSMenu new];

   NSRect winRect;
   NSView *cview;

   NSMutableDictionary *info = [self runDF];
   NSEnumerator *devEnum;
   NSString *device;
   int dcount = 0;

   NSInvocation *inv;
```

The intial setup is straightforward: the menu contains just one entry, namely the item "quit." The width of the window's content view is the same as that of a `PercentageView`. The height of the window is the number of device entries times the height of a single `PercentageView`. We intialize the content rectangle

of the window and allocate the window, set its title, and obtain its content view, to which we'll be adding subviews.

```
[menu addItemWithTitle: @"Quit"
     action:@selector(terminate:)
     keyEquivalent:@"q"];
[NSApp setMainMenu:menu];

winRect = NSMakeRect(0, 0, PWIDTH, PHEIGHT*[info count]);
dfWin = [[NSWindow alloc]
         initWithContentRect:winRect
         styleMask: NSTitledWindowMask
         backing: NSBackingStoreBuffered
         defer: NO];
[dfWin setTitle:@"gdf"];
cview = [dfWin contentView];
```

The next step is important: we initialize the variable `devices`, which contains the dictionary that maps devices to views. Next we obtain an enumerator that iterates over the devices sorted alphabetically, but in reverse order, since we build the subviews with the lowest, i.e. last view first.

```
devices =
   [NSMutableDictionary
      dictionaryWithCapacity:[info count]];
devEnum =
   [[[info allKeys]
       sortedArrayUsingSelector:@selector(compare:)]
      reverseObjectEnumerator];
```

The `while` loop creates the views that go into the window. It stores the array of fields for each device in the variable `items` and allocates the view. The frame rectangle has the standard width and height and is positioned precisely above the previous frame rectangle (we start at the bottom of the window). The titles are created with `infoToTitles`, as in the method `update` that we discussed earlier. We set the percentage once the view has been allocated and the titles have been initialized.

```
while((device = [devEnum nextObject])!=nil){
   NSArray *item = [info objectForKey:device];
   PercentageView *pview =
      [[PercentageView alloc]
         initWithTitles:[self infoToTitles:item]
         andFrame:NSMakeRect(0, dcount*PHEIGHT,
                         PWIDTH, PHEIGHT)];
   [pview setPercentage:
           [[item objectAtIndex:4] intValue]];
```

We record the new view in the dictionary `devices` and add it as a subview to the content view of the window. We increment the counter that determines the vertical position of the current view in the window. We center the window once we are done creating subviews and cause it to be displayed on the screen.

```
   [devices setObject:pview forKey:device];
   [cview addSubview:pview];

   dcount++;
}
[devices retain];

[dfWin center];
[dfWin makeKeyAndOrderFront:nil];
```

It remains to set up the timer. We create an invocation that captures the controller and the method `update` for this purpose. The invocation must persist throughout the lifetime of the application, so we retain it. The last step is to start the timer, which fires once a second.

```
   inv = [NSInvocation
           invocationWithMethodSignature:
              [self methodSignatureForSelector:
                      @selector(update)]];
   [inv setSelector:@selector(update)];
   [inv setTarget:self];
   [inv retain];

   [NSTimer scheduledTimerWithTimeInterval:1
          invocation:inv
          repeats:YES];
}

@end
```

The main routine of this program is very simple. It creates an autorelease pool, an application instance and the controller, which is made the delegate of the application, and starts the application's run loop.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];
   NSApplication *app;

   app = [NSApplication sharedApplication];
   [app setDelegate:[Controller new]];
   [app run];

   [pool release];
   exit(0);
}
```

## 3.3 Code browser with `find` and `grep`

*by Marko Riedel*

### 3.3.1 Idea

This recipe implements a search and find utility with `find` and `grep`. It may be used to search a code base of, say, Objective C code for certain code snippets for use in writing an application. It provides support for coding by "copy-and-paste."

The basic idea is to run `find` on a directory in order to locate a set of files that match a shell pattern or a regular expression. We run `grep` for each file and look for a certain pattern. Matching lines including some context are collected and displayed for viewing. The user may inspect an entire file if she wishes.

We use a subclass of `NSWindow` to provide these features. Every such window consists of three components: the upper component where the user enters the pattern, the regular expression, the shell pattern, the context length and extra switches for `grep` into two forms. The lower half of the window contains the second and third component, which are placed in an `NSSplitView`, namely a scrollview that displays the result of the search and a scrollview for viewing the contents of a match. There is a button between the first and the second component. The user clicks this button to execute the query. It also doubles as a progress indicator during the search, where it fills with blue starting at the left and moving to the right as the search progresses.

The results of a query are displayed with buttons and textviews. The button's title shows what file matched and what line. The textview shows the match including the context. The program displays the entire file in the lower scrollview when the user clicks the button. The context is selected and scrolled so

that it is visible. The query thus produces an alternating sequence of buttons and textviews, one pair for each match.

We need four classes to implement this recipe. The first class is a subclass of `NSView` called `Flipped.` It implements a view whose coordinate system has its origin in the upper left corner, with the y-axis extending downwards. Then there is `RangeButton`, a subclass of `NSButton` that can store a range. It is used in the view that displays the results of the query and it stores the location and the length of the range of a match. There is a controller for interfacing with the application. Finally, the class `BrowseIt` implements the query window described above.

### 3.3.2 Implementation

Start with the usual headers. The class `Flipped` implents a view with a flipped coordinate system. We'll be attaching subviews to this view. It does no drawing itself.

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

@interface Flipped : NSView

- (BOOL)isFlipped;

@end

@implementation Flipped

- (BOOL)isFlipped
{
    return YES;
}

@end
```

The class `RangeButton` is trivial as it only adds an instance variable to store the range and a method to retrieve it.

```
@interface RangeButton : NSButton
{
    NSRange range;
}

- (NSRange)range;
- setRange:(NSRange)aRange;

@end
```

The method `range` reads the instance variable and the method `setRange` writes it.

```
@implementation RangeButton

- (NSRange)range
{
    return range;
}

- setRange:(NSRange)aRange
{
    range = aRange;
    return self;
```

```
}

@end
```

The headers for the controller class and the custom window are next. The controller reads the location of the `find` and `grep` binaries from the user defaults on startup and remembers them during the lifetime of the application. This is the purpose of the two instance variables and the accessors `find` and `grep`. There is an important method that runs a command with some arguments and collects its standard output and standard error streams and returns its exit status. We will be using this method to run `find` and `grep`. There is a method that opens a new query window in response to a click on the corresponding menu item. It lets the user choose the directory to search in an openpanel.

```
@interface Controller : NSObject
{
   NSString *find, *grep;
}

- (NSString *)find;
- (NSString *)grep;

- (int)readFromCommand:(NSString *)cmd
         arguments:(NSMutableArray *)args
           result:(NSArray **)rptr;

- (void)applicationDidFinishLaunching:(NSNotification *)notif;
- open:(id)sender;

@end
```

The following series of definitions pertains to the user interface of the program and to the behavior of the custom window. We will not be displaying tabs in text views and define `TABREP` to be a string of spaces that replace a single tab. We define the width and height of the window with `DIMENSION`. We will be assembling two forms in the upper third of the window and we define an enumerated type that indicates the purpose of a formcell from those two forms. This lets us assemble the forms in a loop instead of coding every cell in turn. We define the titles of the form cells. The last define is for the point size of the fixed pitch font that we will use on buttons and in textviews.

```
#define TABREP @"        "

#define DIMENSION 500

typedef enum {
   ARG_PATTERN = 0,
   ARG_NAME,
   ARG_REGEX,
   ARG_CONTEXT,
   ARG_SWITCHES,
   ARG_COUNT
} ARG;

static NSString *argTitles[ARG_COUNT] = {
   @"Pattern",
   @"Name",
   @"Regex",
   @"Context",
   @"Switches"
};
```

```
#define DISPSIZE 12
```

The custom window contains many instance variables that make it easy to reference the objects in its view hierarchy. It stores the controller because it will use it to find files and grep for patterns. It also remembers what directory it is supposed to process. It stores the formcells that contain the values of the switches for `find` and `grep`. It stores the search button because it will lock focus on it to draw the progress indicator. It also stores the split view and the upper scrollview, which will hold the query results, and the lower scrollview, which is used to view entire files. The query results will be attached to the document view of the upper scrollview as they come in and the variables `attachAtY` and `maxwidth` tell us where they go (what height) and what the maximum width of a result is, respectively. We'll be computing the necessary widths and heights of text views and what rectangle we need to scroll to for display of a match in the file viewer. That's why we store the fixed pitch font and its bounding box.

```
@interface BrowseIt : NSWindow
{
   Controller *con;

   NSString *directory;

   NSFormCell *args[ARG_COUNT];
   NSButton *sbutton;
   NSSplitView *split;
   NSScrollView *scrollUpper, *scrollLower;

   float attachAtY, maxwidth;

   NSFont *sfont;
   NSSize bbox;
}
```

A custom window (we will also refer to it by the term "browse window") is initialized with the value of the directory that we will search and the controller that provides the facility to run commands and collect the output. There are two shorthand methods to set the document view of the upper and lower scrollviews.

```
- initWithDirectory:(NSString *)dir
       controller:(Controller *)theController;

- setUpper:(id)uv;
- setLower:(id)lv;
```

We store the parameters of the last successful query in the user defaults, and read them back in for use as the initial values when we open a new browse window. This is the purpose of `readArgs` and `writeArgs`.

```
- readArgs;
- writeArgs;
```

The browse window will be the delegate of the split view that it contains. We implement two methods that prevent the user from completely miniaturizing the upper or the lower scrollview.

```
- (float)splitView:(NSSplitView *)sender
constrainMinCoordinate:(float)proposedMin ofSubviewAt:(int)offset;
- (float)splitView:(NSSplitView *)sender
constrainMaxCoordinate:(float)proposedMax ofSubviewAt:(int)offset;
```

We have a method that takes the output lines of a single successful grep on a single file and assembles the corresponding series of buttons and textviews.

```
- processMatchFor:(NSString *)fname data:(NSArray *)lines;
```

There is a method to display the contents of the file represented by a button in the lower scrollview.

```
- loadIt:(id)sender;
```

We will respond to a certain number of error conditions. The method `errView` returns a view that holds the error message and can be placed in one of the two scrollviews.

```
- (NSTextView *)errView:(NSString *)msg;
```

The method `search:` is the action of the search button. It runs `find` and iterates over the files that it outputs, invoking first `grep` and then `processMatchFor:data:` for each file in turn.

```
- search:(id)sender;
```

There is a deallocation method that frees the directory string. Deallocation of views will be handled by placing them in an autorelease pool upon creation, so that they are freed when they are removed from the view hierarchy or when the window is closed.

```
- (void)dealloc;

@end
```

We discuss the implementation of the controller before the implementation of the browse window. There are two accessors that retrieve the location of the `find` and `grep` commands, which was set on startup.

```
@implementation Controller

- (NSString *)find
{
   return find;
}

- (NSString *)grep
{
   return grep;
}
```

We now discuss the very important method that runs tasks and collects their output and error streams. It returns two arrays of lines by reference in the variable `rptr`. The first step is to create a task object and set its launch path and its arguments. (The call to `NSLog` can aid in debugging.)

```
- (int)readFromCommand:(NSString *)cmd
         arguments:(NSMutableArray *)args
           result:(NSArray **)rptr
{
   NSTask *aTask = [NSTask new];

   [aTask setLaunchPath:cmd];
   [aTask setArguments:args];

   // NSLog(@"%@ %@", cmd, args);
```

We need a pipe to the output and error streams, so we create the appropriate objects. We retrieve two file handles for reading from the two pipes.

```
   NSPipe
      *outPipe = [NSPipe pipe],
      *errPipe = [NSPipe pipe];
   NSFileHandle
      *outReader = [outPipe fileHandleForReading],
      *errReader = [errPipe fileHandleForReading];
```

We are now almost ready to lauch the task. We connect the two pipes to the two streams and declare the variables that will hold the data that they produce.

```
[aTask setStandardOutput:outPipe];
[aTask setStandardError:errPipe];

NSData *outData, *errData;
```

The actual run of the task takes place inside an exception handler. We try to run the task and read the data from the two output streams. We wait until the task exits and close the file descriptors that we used to read data from the task.

```
NS_DURING

[aTask launch];
outData = [outReader readDataToEndOfFile];
errData = [errReader readDataToEndOfFile];
[aTask waitUntilExit];

close([outReader fileDescriptor]);
close([errReader fileDescriptor]);
```

The error handler terminates the task if something went wrong and the task is still running. It sets the two arrays of output and error lines to contain the reason for the exception and returns $-1$.

```
NS_HANDLER
if([aTask isRunning]==YES){
   [aTask terminate];
}

close([outReader fileDescriptor]);
close([errReader fileDescriptor]);

NSString *reason = [localException reason];
NSArray *ret = [NSArray arrayWithObject:reason];

rptr[0] = ret;
rptr[1] = ret;
return -1;
NS_ENDHANDLER
```

What remains will only be executed if we were successful in launching the task and reading from its two data streams. We check the two data objects for the presence of data. If there are data, then we convert them into a C string and split this string into lines. We do not include the last return character because it would produce an empty string at the end of the array.

```
int p;
for(p = 0; p<2; p++){
   NSData *data = (p>0 ? errData : outData);
   rptr[p] =
      (![data length] ? [NSArray array] :
       [[NSString stringWithCString:[data bytes]
                length:[data length]-1]
          componentsSeparatedByString:@"\n"]);
}
```

The method returns the exit status of the task, which is an important value that can tell us whether the task succeeded or not and what problems there were, if any.

```
    return [aTask terminationStatus];
}
```

This almost completes the implementation of the controller. The penultimate method is invoked when the application finishes launching and it reads the location of the two binaries for finding and grepping from the user defaults. It requests the defaults object and the file manager for this purpose.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
    NSUserDefaults *ud =
        [NSUserDefaults standardUserDefaults];
    NSFileManager *fm =
        [NSFileManager defaultManager];
```

It looks for the find binary in the defaults and sets it to a default value if there was no entry. It raises an exception if the binary is not executable.

```
    if((find = [ud stringForKey:@"find"])==nil){
        find = @"/usr/bin/find";
    }
    if([fm isExecutableFileAtPath:find]==NO){
        [NSException raise:NSGenericException
                    format:@"bad find binary: %@", find];
    }
    [find retain];
```

The grep binary is handled the same way: look for it among the defaults, assign a default value if it is not found, and check that it is executable.

```
    if((grep = [ud stringForKey:@"grep"])==nil){
        grep = @"/usr/bin/grep";
    }
    if([fm isExecutableFileAtPath:grep]==NO){
        [NSException raise:NSGenericException
                    format:@"bad grep binary: %@", grep];
    }
    [grep retain];
}
```

The last method responds to an entry on the application's main menu and lets the user choose the directory that she wants to search. It runs a pretty standard openpanel dialogue to get this value. It obtains the openpanel and sets the title. The user may choose directories but not files and no multiple selections are allowed.

```
- open:(id)sender
{
    NSOpenPanel *openPanel = [NSOpenPanel openPanel];

    [openPanel setTitle:@"Open directory"];
    [openPanel setAllowsMultipleSelection:NO];
    [openPanel setCanChooseDirectories:YES];
    [openPanel setCanChooseFiles:NO];
```

Should there be an entry for the key "Directory" among the user defaults, then we try to open this directory. (The value for this key is updated after successful queries.) We use the current directory if there was no entry.

```
    NSFileManager *fm = [NSFileManager defaultManager];
    NSString *dir =
```

```
      [[NSUserDefaults standardUserDefaults]
         stringForKey:@"Directory"];
   if(dir==nil){
      dir = [fm currentDirectoryPath];
   }
```

We only set the openpanel's directory if it is indeed a directory of the file system.

```
   BOOL isDir;
   if([fm fileExistsAtPath:dir isDirectory:&isDir]==YES &&
     isDir==YES){
      [openPanel setDirectory:dir];
   }
```

The last step is to run the panel. We create a new browse window for the chosen directory if the user clicked the okay button. We center the window on the screen and order it to the front.

```
   if([openPanel runModalForTypes:nil]==NSOKButton){
      BrowseIt *bwin =
         [[BrowseIt alloc]
            initWithDirectory:[openPanel filename]
            controller:self];
      [bwin center];
      [bwin makeKeyAndOrderFront:self];
   }

   return self;
}

@end
```

We may now discuss the implementation of the browse window. The initializer is first. It is very simple conceptually, since it only needs to assemble the views that go into the content view of the window. The first step is to store and retain the directory. The controller is also stored but does not need to be retained. Next we declare and set the frame of the new window and its style mask, making it closable, titled, and resizable.

```
@implementation BrowseIt

- initWithDirectory:(NSString *)dir
       controller:(Controller *)theController
{
   directory = dir;
   [directory retain];

   con = theController;

   NSRect wframe =
      NSMakeRect(0, 0, DIMENSION, DIMENSION);
   int mask = NSTitledWindowMask | NSClosableWindowMask |
      NSResizableWindowMask;
   [super
      initWithContentRect:wframe
      styleMask:mask
      backing:NSBackingStoreBuffered
      defer:NO];
```

We set the minimum size of the window and its title, which contains the name of the directory being searched.

```
[self setMinSize:wframe.size];
[self setTitle:
        [NSString
           stringWithFormat:@"browse %@",
           directory]];
```

We must assemble the subviews that make up the interface. Start with the two forms in the upper third. We declare the frame for the forms and initialize it to be the right width; the height and the origin are set later. We declare `arg` to iterate over the form entries (the parameters) that we declared earlier. The variable `f` will be used later to iterate over the two forms, left and right, that is.

```
NSRect fframe =
   NSMakeRect(0, 0, DIMENSION, 0);
NSForm *form[2];

form[0] = [[NSForm alloc] initWithFrame:wframe];
form[1] = [[NSForm alloc] initWithFrame:wframe];

int arg, f;
```

We iterate over the arguments that we require and place them in the two forms, reading the statically declared titles and choosing the left form for the first half and the right form for the second.

```
for(arg=0; arg<ARG_COUNT; arg++){
   args[arg] =
      [form[arg<=ARG_COUNT/2 ? 0 : 1]
          addEntry:[NSString
                    stringWithFormat:
                       @"%@:", argTitles[arg]]];
}
```

It remains to customize the forms and their cells to serve our needs. We ask the first cell for the size of the font that it uses and obtain a fixed pitch font of this size.

```
NSFormCell *cell = [form[0] cellAtIndex:0];
float psize = [[cell font] pointSize];
NSFont *fixed = [NSFont userFixedPitchFontOfSize:psize];
```

Now we iterate over the two forms, setting the label and the content font first.

```
for(f=0; f<2; f++){
   [form[f] setTextFont:fixed];
   [form[f] setTitleFont:fixed];
```

The titles are supposed to be right justified. A single cell is as wide as half the window and ten points higher than its font.

```
   [form[f] setTitleAlignment:NSRightTextAlignment];

   [form[f] setCellSize:NSMakeSize(DIMENSION/2, psize+10)];
```

We move each form to the right location. The origin of the left form is at the window's border and the right form is next to the left. The upper boundary of the form coincides with the upper boundary of the window.

```
   fframe.origin.x = f*DIMENSION/2;
   fframe.origin.y = DIMENSION-[form[f] frame].size.height;
   [form[f] setFrameOrigin:fframe.origin];
```

The forms are widthsizable. The left form is attached to the left border of the window, and the right form to the right.

```
    mask = NSViewWidthSizable | NSViewMinYMargin |
       (f>0 ? NSViewMinXMargin : NSViewMaxXMargin);
    [form[f] setAutoresizingMask:mask];
```

The forms are made subviews of the window's content view and put in an autorelease pool.

```
    [[self contentView] addSubview:form[f]];
    [form[f] autorelease];
}
```

We connect the left form to the right so that the user may tab through the fields.

```
[form[0] setNextKeyView:form[1]];
```

The search button lies below the two forms and is as wide as the window. We initialize its frame rectangle and allocate the button.

```
NSRect bframe =
   NSMakeRect(0, 0, DIMENSION, psize+10);
sbutton =
   [[NSButton alloc] initWithFrame:bframe];
```

The title of the button is "search" and it gets the same font as the formcells. The window is its target and the action that it triggers when clicked is `search:`.

```
[sbutton setTitle:@"Search"];
[sbutton setFont:fixed];
[sbutton setTarget:self];
[sbutton setAction:@selector(search:)];
```

We wish to obtain the appropriate height of the button and move it to its place below the two forms. Hence we ask it for the appropriate size, of which we use the value for the height. The origin is below the two forms. We set the button's frame.

```
[sbutton sizeToFit];

bframe = [sbutton frame];
bframe =
   NSMakeRect(0, [form[0] frame].origin.y−
           bframe.size.height,
           DIMENSION, bframe.size.height);

[sbutton setFrame:bframe];
```

The button is widthsizable and remains attached to the upper boundary of the window.

```
mask = NSViewWidthSizable | NSViewMinYMargin;
[sbutton setAutoresizingMask:mask];
```

The button becomes the key view should the user tab out of the last field of the right form. Tabbing again takes the user back to the first field of the left form, which is the initial first responder of the window.

```
[form[1] setNextKeyView:sbutton];
[sbutton setNextKeyView:form[0]];

[self setInitialFirstResponder:form[0]];
```

We attach the button to the window's content view.

```
[[self contentView] addSubview:sbutton];
```

The lower part of the window is occupied by a split view, which we now create. This view will hold two scrollviews and occupies the space left over after the forms and the button have been taken into account. Its delegate is the window itself, which implements size restraint messages during resize operations.

```
NSRect spframe =
   NSMakeRect(0, 0, DIMENSION, bframe.origin.y);
split = [[NSSplitView alloc] initWithFrame:spframe];
[split setDelegate:self];
```

The splitview is widthsizable and heightsizable. It expands to fill the entire lower part of the window. We attach it to the view hierarchy.

```
[split setAutoresizingMask:
        NSViewWidthSizable | NSViewHeightSizable];
[[self contentView] addSubview:split];
```

It remains to create the two scrollviews. Both are half as high as the split view and as wide as the window. The upper scrollview is placed in the upper half of the splitview.

```
NSRect scframe =
   NSMakeRect(0, spframe.size.height/2,
            DIMENSION, spframe.size.height/2);
scrollUpper = [[NSScrollView alloc] initWithFrame:scframe];
```

It should resize with the splitview, has a vertical scroller and its background is white. We place it in the splitview.

```
[scrollUpper
   setAutoresizingMask:
      NSViewWidthSizable | NSViewHeightSizable];

[scrollUpper setHasVerticalScroller:YES];
[scrollUpper setBackgroundColor:[NSColor whiteColor]];

[split addSubview:scrollUpper];
```

The lower scrollview has the same dimensions as the upper one, but is placed in the lower left corner of the splitview.

```
scframe =
   NSMakeRect(0, 0,
            DIMENSION, spframe.size.height/2);
scrollLower = [[NSScrollView alloc] initWithFrame:scframe];
```

Its sizing behavior is the same as that of the upper scrollview, as are its choice of scrollers and its background color. It is also placed in the view hierarchy.

```
[scrollLower
   setAutoresizingMask:
      NSViewWidthSizable | NSViewHeightSizable];

[scrollLower setHasVerticalScroller:YES];
[scrollLower setBackgroundColor:[NSColor whiteColor]];

[split addSubview:scrollLower];
```

All of these views go into an autorelease pool, so that they are freed upon removal from the view hierarchy and upon closure of the window.

```
   [split autorelease];
   [sbutton autorelease];
   [scrollUpper autorelease];
   [scrollLower autorelease];
```

We are now done assembling the window and read the arguments of the last successful query, if any, into the formcells of the two forms.

```
   [self readArgs];
```

We select the text in the first cell so that the user can immediately start typing when the window appears.

```
   [form[0] selectTextAtIndex:0];
```

We'll be using a fixed pitch font for the textviews that display matches. We now obtain a font of the right size and the dimensions of its bounding box for later use. This ends the initializer.

```
   sfont = [NSFont userFixedPitchFontOfSize:DISPSIZE];
   bbox.width = [sfont boundingRectForFont].size.width;
   bbox.height = [sfont defaultLineHeightForFont];

   return self;
}
```

New document views of the two scrollviews need to be autoreleased so that they are later freed at the appropriate times. We define two methods that send an autorelease message to a view before placing it in the upper or lower scrollview.

```
- setUpper:(id)uv
{
   TEST_AUTORELEASE(uv);
   [scrollUpper
      setDocumentView:uv];
   return self;
}

- setLower:(id)lv
{
   TEST_AUTORELEASE(lv);
   [scrollLower
      setDocumentView:lv];
   return self;
}
```

The process of reading the search parameters from the user defaults is very simple. First obtain the defaults object, then retrieve the values by their titles and write them into the appropriate form cell.

```
- readArgs
{
   NSUserDefaults *ud =
      [NSUserDefaults standardUserDefaults];

   int p;
   for(p=0; p<ARG_COUNT; p++){
      [args[p] setStringValue:
             [ud stringForKey:argTitles[p]]];
   }

   return self;
```

```
}
```

The search parameters are written to the user defaults after every successful search. This is done by first retrieving the user defaults.

```
- writeArgs
{
   NSUserDefaults *ud =
      [NSUserDefaults standardUserDefaults];
```

Then we iterate over the cells that hold the parameters, checking their values in turn. Values that are not empty are recorded in the user defaults.

```
   int p;
   for(p=0; p<ARG_COUNT; p++){
      NSString *val = [args[p] stringValue];
      if(val!=nil){
         [ud setObject:val
            forKey:argTitles[p]];
      }
   }
```

We also record the current directory. The write process ends with an invocation of `synchronize`, so that the parameter values will be used the next time a new window is opened.

```
   [ud setObject:directory forKey:@"Directory"];

   [ud synchronize];

   return self;
}
```

Recall that the browse window is the splitview's delegate. We implement two methods that the split view will invoke when the divider is moved; one to constrain the minumum coordinate and one the maximum coordinate. We set these to be a sixth of the splitview's height away from its lower and upper boundary respectively. This keeps the user from moving the divider to a position where the upper or lower scrollview can no longer be scrolled in a useful way.

```
- (float)splitView:(NSSplitView *)sender
constrainMinCoordinate:(float)proposedMin ofSubviewAt:(int)offset
{
   return [sender frame].size.height/6;
}

- (float)splitView:(NSSplitView *)sender
constrainMaxCoordinate:(float)proposedMax ofSubviewAt:(int)offset
{
   return [sender frame].size.height*5/6;
}
```

The method `processMatchFor:data:` plays a key role in the application. It is invoked when the grep was successful and the file contained lines that match the pattern. Here is an example of the output from the grep command.

```
grep -n -C 1 dealloc tasks/browseit/code/browseit.m
102-
103:- (void)dealloc;
104-
--
```

```
844-
845:- (void)dealloc
846-{
847-    [directory release];
848:    [super dealloc];
849-}
```

The input of this method is the name of the file and the output lines. A line either contains match and context data or it is a divider between sections. We skip those dividers. We read each line in turn and produce a button and a textview for the section whenever the line number of the current line is not one more than the previous line number, which is how we recognize sections. We start by getting the document view of the upper scrollview. We'll attach sections to this view. This method merely attaches sections at the bottom edge of the previous section, which is stored in the instance variable `attachAtY`. The caller is responsible for resizing the document view once all sections have been attached. We declare variables for iterating over the output lines. Actually our loop will include an extra iteration (a divider) at the very end, which is how we process the last section. We will store the range of the current section, i.e. where it begins and how many lines it contains. We store the lines of each section in a mutable data object. We will also compute the length of the longest line (`maxlen`) of each section so that we may choose the appropriate width of the textview.

```
- processMatchFor:(NSString *)fname data:(NSArray *)lines
{
   NSView *docView = [scrollUpper documentView];
   int l, m = [lines count];

   NSRange range = NSMakeRange(-1, 0);
   NSMutableArray *data;

   int maxlen, curlen;
```

We iterate over the lines including an extra iteration at the end, where we use a divider as the line's content. We extract the current line, replace tabs by spaces and convert it into a C string.

```
   for(l=0; l<m+1; l++){
      NSString *line =
         (l<m ? [lines objectAtIndex:l] : @"--");
      const char *str =
         [[line stringByReplacingString:@"\t"
               withString:TABREP] UTF8String];
```

Recall that the line number starts the line. We extract its value and skip over the digits by which it is represented. If there were no digits, then we have a divider, and we skip all dividers except for the last one.

```
      int lnumber = atoi(str), digits = 0;
      while(*str && isdigit(*str)){
         str++; digits++;
      }

      if(!digits && l<m){
         continue;
      }
```

The actual content of the line is stored in the variable `rest`. Note that we must skip over the colon that marks matches and the dash that marks context. We have a new section if we are at the very end or if there was a skip in the line number.

```
      NSString *rest =
         [NSString stringWithCString:str+1];
      curlen = [rest length];
```

```
    if(l==m || lnumber > range.location+range.length){
```

First we create the textview that holds the match and its context. We compute its size from the longest line (width) and the total number of lines in this section (height). We allocate it and initialize it with the right frame. We will set the origin later. It should go below the previously attached views.

```
        // text

        NSSize textbsize =
           NSMakeSize(maxlen*bbox.width,
                   [data count]*bbox.height);

        NSTextView *tv =
           [[NSTextView alloc]
              initWithFrame:
                 NSMakeRect(0, 0,
                         textbsize.width,
                         textbsize.height)];
```

We set it not to be editable, to use the fixed pitch font, and to contain the current set of lines and put it into an autorelease pool for automatic deallocation upon removal from the view hierarchy or a window closure.

```
        [tv setEditable:NO];
        [tv setFont:sfont];
        [tv setString:[data componentsJoinedByString:@"\n"]];
        [tv autorelease];
```

Next we create the button that will take the user to the entire file if clicked. It is situated above the textview. We will adjust the dimensions and the origin of the button later.

```
        // button

        RangeButton *rb =
           [[RangeButton alloc]
              initWithFrame:NSMakeRect(0,
                                textbsize.height,
                                50, 50)];
```

The button's title lists the name of the file and the line number. The range that it represents is the current range. It is a momentary push button.

```
        [rb setTitle:
              [NSString stringWithFormat:@"%@: %d",
                     fname, (int)range.location]];
        [rb setButtonType:NSMomentaryPushButton];
        [rb setRange:range];
```

The window i.e. `self` is the target of the button and its action is the method that loads a file into the lower scrollview. The button represents the current file, which is relative to the directory that is stored in the corresponding instance variable.

```
        [rb setTarget:self];
        [rb setAction:@selector(loadIt:)];
        [[rb cell] setRepresentedObject:fname];
```

We are just about done with the button and ask it to set its frame to its preferred size. The button also goes into an autorelease pool.

```
        [rb sizeToFit];
        NSSize btnsize = [rb frame].size;

        [rb autorelease];
```

The instance variable `maxwidth` stores the maximum width of the buttons and textviews created during the current find process. It must be updated if the new button or textview is wider than its value. We build a chain of views moving downwards along the y-axis of the document view and attach the new sections at the bottom of the chain.

```
        if(maxwidth<btnsize.width){
           maxwidth = btnsize.width;
        }
        if(maxwidth<textbsize.width){
           maxwidth = textbsize.width;
        }
```

We attach the button and the textview at the bottom of the chain. We also move the attachement point down by the combined height of the new objects. This completes the construction of a new section.

```
        [rb setFrameOrigin:
              NSMakePoint(0, attachAtY)];
        [tv setFrameOrigin:
              NSMakePoint(0, attachAtY+btnsize.height)];

        attachAtY += btnsize.height+textbsize.height;

        [docView addSubview:tv];
        [docView addSubview:rb];
     }
```

Recall that we store state as we iterate over the output lines, namely the contents of the current section and its range. We must reset these values at the beginning and after a section has been completed. These are the two cases that the following `if` statement detects. We reset the section line array to the current line, set the range to be a one-line range at the current line and the maximum number of characters to be those of the current line.

```
     if(!l ||
       (l<m && lnumber > range.location+range.length)){
        data = [NSMutableArray arrayWithObject:rest];
        range = NSMakeRange(lnumber, 1);
        maxlen = curlen;
     }
```

There is also housekeeping to do when we are inside a section. We add the current line to the array of lines, increase the length of the range by one and update the maximum width in characters if necessary. This concludes the processing method, which is invoked with the grep output for every file that matches the name or regular expression given in the form.

```
     else{
        [data addObject:rest];
        range.length++;
        if(curlen>maxlen){
           maxlen = curlen;
        }
     }
  }
```

```
    return self;
}
```

We have seen that the buttons for each section have the browse window as the target. A click on one of these buttons should load the entire file that the section excerpts and scroll to the excerpt. The method `loadIt:` is the action of these buttons and implements the desired load behavior. It starts by computing the full path to the file and attempts to load it into memory (into a string object).

```
- loadIt:(id)sender
{
   NSString *fullName =
      [directory
         stringByAppendingPathComponent:
            [[sender cell] representedObject]];
   NSString *contents =
      [NSString stringWithContentsOfFile:fullName];
```

A descriptive error message will be displayed in the lower scrollview if the file cannot be loaded. We replace all tabs by spaces if the file did load correctly.

```
   if(contents==nil){
      NSString *estr =
         [NSString
            stringWithFormat:@"Couldn't read: \"%@\"",
            fullName];

      [self setLower:[self errView:estr]];
      return self;
   }

   NSString *noTabs =
      [contents stringByReplacingString:@"\t"
              withString:TABREP];
```

We now prepare to iterate over the characters in the file in order to compute several statistics. We need to transform the range that is to be selected from a range of lines into a range of characters. We also need to know the widest line of the file so that we may choose the right size of textview. The variable `ptr` is used to iterate over the string and `prev` holds the offset in characters of the previous line. The variables `from` and `to` describe the extent of the selection.

```
   const char *str = [noTabs UTF8String],
      *ptr = str, *prev = str;
   int lines = 0, curlen = 0, maxlen = -1;

   NSRange lineRange = [sender range], selRange;
   int
      from = lineRange.location,
      to = lineRange.location+lineRange.length-1;
```

Start iterating over the characters. We have a line feed if we see a newline character or if we are at the end of the string, in which case the length of the line should include the last character.

```
   while(*ptr){
      if(*ptr=='\n' || !ptr[1]){
         lines++;
         if(!ptr[1]){
            curlen++;
         }
```

If the length of the current line is larger than the recorded maximum, then the maximum is updated.

```
        if(curlen>maxlen){
           maxlen = curlen;
        }
        curlen = 0;
```

We record the character offset if we have found the first line. It starts one character after the previous newline. Similarly, the length of the range in characters is the difference between the current offset and the start of the range. The current newline becomes the previous line now that we are done.

```
        if(lines==from){
           selRange.location = 1+prev-str;
        }
        if(lines==to){
           selRange.length =
              (ptr-str) - selRange.location;
        }
        prev = ptr;
```

Characters in the interior of a line merely increase the length of the line. We move to the next character in all cases.

```
     else{
        curlen++;
     }
     ptr++;
  }
```

We may create the textview once all relevant statistics have been computed. Its dimensions are determined by the length of the widest line and the total number of lines. We allocate the view and initialize it with the right frame size.

```
  NSSize textbsize =
     NSMakeSize(maxlen*bbox.width, lines*bbox.height);

  NSTextView *tv =
     [[NSTextView alloc]
        initWithFrame:
           NSMakeRect(0, 0,
                   textbsize.width,
                   textbsize.height)];
```

The view is not editable and uses the fixed pitch font that is also used in the section view. Its contents are the contents of the file with tabs replaced by spaces. We select the lines that were shown in the section above and place the new textview in the lower scrollview.

```
  [tv setEditable:NO];
  [tv setFont:sfont];
  [tv setString:noTabs];
  [tv setSelectedRange:selRange];

  [self setLower:tv];
```

We now compute the rectangle that should be visible in the scrollview. It is given by the upper part of the selected range. We could have used `scrollRangeToVisible:` for this purpose, but its current implementation only shows the first line, and we wish to show as many lines as possible. We compute the visible height of the content view of the lower scrollview, minus one line. (This really is the content view and not the document view.) We compute the rectangle `vrect`, which encloses the selection. If the rectangle

is higher than what can be displayed, then its height is set to the maximum possible value. Finally we ask the textview to scroll the rectangle into the visible part of the scrollview. This completes the method that loads files in response to button clicks on range buttons.

```
   float visheight =
      [[scrollLower contentView] frame].size.height
      −bbox.height;
   NSRect vrect =
      NSMakeRect(0, from∗bbox.height,
              0, (to−from)∗bbox.height);
   if(vrect.size.height>visheight){
      vrect.size.height = visheight;
   }
   [tv scrollRectToVisible:vrect];

   return self;
}
```

We have already encountered the method `errView:` on several occasions. Recall that it manufactures a textview containing an error message for placement in the upper or lower scrollview. This method is kept simple and can certainly be improved. We make a textview whose frame has the default width in use throughout the application and is half as high as this default. We choose an eighteen-point system font for our messages.

```
− (NSTextView ∗)errView:(NSString ∗)msg
{
   #define FSIZE 18
   NSTextView ∗ev =
      [[NSTextView alloc]
         initWithFrame:
            NSMakeRect(0, 0,
                    DIMENSION, DIMENSION/2)];
```

Error messages are of course not editable and we record this fact. The background color of the view is white and it uses the font mentioned above.

```
   [ev setFont:[NSFont systemFontOfSize:FSIZE]];
   [ev setBackgroundColor:[NSColor whiteColor]];
   [ev setEditable:NO];
```

We prefix the error message with the current date and time. This is so that the message changes even if the user repeatedly clicked the search button or a section button. The change in the date shows the user that processing has occurred.

```
   NSString ∗dateStr =
      [[NSDate date]
         descriptionWithCalendarFormat:@"%a,␣%d−%b−%Y␣"
         @"%H:%M:%S"
         timeZone:[NSTimeZone localTimeZone]
         locale:nil],
      ∗estr =
      [NSString stringWithFormat:@"%@:\n%@", dateStr, msg];
```

We store the error message with the prefix in the textview and return the result.

```
   [ev setString:estr];
   return ev;
}
```

The penultimate method of a browse window and of this program is the action method `search:`. It invokes `find` and runs `grep` on all matching files, using `processMatchFor:data:` to assemble the button-textview pairs that go into the upper scrollview. Start by constructing the arguments to `find`. We indicate that `find` should follow symbolic links and list only files and not directories.

```
- search:(id)sender
{
   NSMutableArray *findArgs =
      [NSMutableArray
         arrayWithObjects: directory,
         @"-follow", @"-type", @"f", nil];
```

We read the values for the name and the regular expression from the form cells. (Note that the name applies to the name of the file and the regular expression applies to the whole path.) We add the name pattern if there was one. Similarly, if the user entered a regular expression, then it is added to the find arguments.

```
   NSString
      *name = [args[ARG_NAME] stringValue],
      *regex = [args[ARG_REGEX] stringValue];

   if([name length]>0){
      [findArgs addObject:@"-name"];
      [findArgs addObject:name];
   }
   if([regex length]>0){
      [findArgs addObject:@"-regex"];
      [findArgs addObject:regex];
   }
```

We must specify an action for find to take on matching files. We choose the action `printf` with the directive `P`, which prints the file's "name with the name of the command line argument under which it was found removed," so that the user sees path components starting with the components in the directory that is being searched.

```
   [findArgs addObject:@"-printf"];
   [findArgs addObject:@"%P\n"];
```

We are now ready to invoke `find`. We ask the controller to run the find program and collect its standard output and standard error streams in two arrays. The array `files` contains the standard output.

```
   NSArray *data[2], *files;
   int res =
      [con readFromCommand:[con find]
          arguments:findArgs result:data];
   files = data[0];
```

There are two possible error conditions. The first occurs if `find` exited with a non-zero exit code, in which case we collect its standard error for display with `errView:`. It is also an error if `find` did not find any files. The error message goes into the upper scrollview in either of these cases.

```
   NSString *estr = nil;
   if(res){
      estr = [data[1] componentsJoinedByString:@"\n"];
   }
   else if(![files count]){
      estr = @"no_files_found";
   }
   if(estr!=nil){
```

```
    [self setUpper:[self errView:estr]];
    [self setLower:nil];
    return self;
}
```

We sort the files if there was no error.

```
NSArray *sorted =
    [files sortedArrayUsingSelector:
            @selector(caseInsensitiveCompare:)];
```

We now prepare the arguments to `grep`. Only the last argument (the file being searched) will change as we iterate over the files; all the other arguments stay constant. The first argument is absolutely essential: it tells `grep` to include line numbers in its output, which we use to parse it into sections. If the user entered a context value (number of lines surrounding a match that should be displayed), then this value is used.

```
NSMutableArray *grepArgs =
    [NSMutableArray arrayWithObject:@"-n"];

NSString *context = [args[ARG_CONTEXT] stringValue];
if([context length]){
    [grepArgs addObject:@"-C"];
    [grepArgs addObject:context];
}
```

We also have a form cell for additional switches that can be passed on to `grep`, like the switch `-i` for case-insenstive matching or `-E` for extended regular expressions (very useful). We read the value from the cell and split it on the space character; all switches that are not empty are then added to the arguments for `grep`.

```
NSEnumerator *switchEnum =
    [[[args[ARG_SWITCHES] stringValue]
        componentsSeparatedByString:@" "]
       objectEnumerator];
NSString *switx;
while((switx=[switchEnum nextObject])!=nil){
    if([switx length]){
        [grepArgs addObject:switx];
    }
}
```

We create the document view of the upper scrollview before we start the loop. Its dimensions will be set later, once all sections have been computed. We allocate and initialize it and put it into the scrollview. The section buttons and textviews will be attached to this view.

```
NSRect initialFrame =
    NSMakeRect(0, 0, DIMENSION, 0);
NSView *pv = [[Flipped alloc] initWithFrame:initialFrame];
[self setUpper:pv];
```

The last argument to `grep` that does not change is the pattern.

```
[grepArgs addObject:[args[ARG_PATTERN] stringValue]];
```

Recall that `grep` signals when it finds a match in a binary file. We collect these matches into an array that we display for the user when there were no matches in text files.

```
NSMutableArray *binaries =
    [NSMutableArray arrayWithCapacity:1];
```

We may now start iterating over the file names. The variable `fpos` indicates the current position and the variable `fmax` the number of files. We set `attachAtY` to be the bottom margin of the document view and the maximum width of a section to be zero. Recall that we must lock focus on the search button because we will use it as a progress indicator, filling it with blue starting at the left and moving to the right as we process files.

```
unsigned int fpos, fmax = [files count];
unsigned matchCount = 0;

NSRect sframe = [sbutton frame];
[sbutton lockFocus];
[[NSColor blueColor] set];

attachAtY = 0; maxwidth = 0;
```

The first thing we do inside the loop is to draw the progress indicator. The ratio `fpos/fmax` indicates what width to fill. We draw a rectangle and flush the window so that the user sees the indicator advance. We create an autorelease pool that will hold objects created during processing, so that we can immediately release those that are no longer referenced after an item has been processed, as opposed to releasing them in the application's run loop.

```
for(fpos=0; fpos<fmax; fpos++){
   NSRect toFill =
      NSMakeRect(0, 0,
              sframe.size.width*fpos/fmax,
              sframe.size.height);
   NSRectFill(toFill);
   [self flushWindow];

   NSAutoreleasePool *pool = [NSAutoreleasePool new];
```

The last argument to `grep`, which is not constant, is the full path to te file being searched. We ask the directory string to provide the path that is obtained by appending the relative path to the file.

```
   NSString *name = [sorted objectAtIndex:fpos];

   [grepArgs
      addObject:
         [directory
            stringByAppendingPathComponent:name]];
```

We run `grep` with the array of arguments and extract the contents of the standard output stream.

```
   res = [con readFromCommand:[con grep]
           arguments:grepArgs result:data];
   NSArray *grepped = data[0];
```

The exit status of `grep` is 2 if an error occurred. If this is the case, then we build an error message that contains the name of the file and the error message from `grep`, or rather, the contents of its standard error stream. This error message is placed in the upper scrollview. We must unlock focus from the search button and restore it to its former state before we exit the routine.

```
   if(res==2){
      estr =
         [NSString
            stringWithFormat:@"File is: \"%@\"\n%@", name,
            [data[1] componentsJoinedByString:@"\n"]];

      [self setUpper:[self errView:estr]];
```

```
        [self setLower:nil];

        [sbutton unlockFocus];
        [sbutton display];

        return self;
    }
```

The exit status of `grep` is zero if there was a match. There are two possibilities: there was a text file match or a match in a binary file. The former includes match and context data that are prefixed with line numbers; the latter, a message that there was a match. Hence we may distinguish the two cases by checking if the first line begins with a digit. If so, we process the output into sections, assembling buttons and textviews with `processMatchFor:data:`. If not, the file is added to the array of binaries.

```
    else if(!res){
        // skip binary data
        const char *first =
            [[grepped objectAtIndex:0] UTF8String];

        if(isdigit(*first)){
            [self processMatchFor:name data:grepped];
            matchCount++;
        }
        else{
            [binaries addObject:name];
        }
    }
```

The loop ends with the removal of the file name from the arguments; i.e. we pop the name that we pushed at the end of the array at the beginning of the loop. We also release memory that was allocated during processing of the current file and is no longer used. We unlock focus and restore the search button's appearance once we are done processing all the files.

```
    [grepArgs removeLastObject];

    [pool release];
}

[sbutton unlockFocus];
[sbutton display];
```

It is quite possible that there were no matches. We must inform the user of this event. We produce an error message to this effect. It includes the list of binary matches if there were any and goes into the upper scrollview.

```
if(!matchCount){
    estr = @"no text matches found";
    if([binaries count]){
        NSString
            *blist =
            [binaries componentsJoinedByString:@"\n"],
            *fmt = @"\nbinary matches:\n%@";
        estr =
            [estr stringByAppendingFormat:fmt, blist];
    }

    [self setUpper:[self errView:estr]];
    [self setLower:nil];
```

```
    }
```

There is some clean-up work to do if we did find matches. We now know the required total size of the document view and resize it accordingly. We tell the document view to scroll to the first match, so that the user sees the first file that matched in the upper left corner of the scrollview's content view.

```
    else{
        [pv setFrameSize:
              NSMakeSize(maxwidth, attachAtY)];
        [pv scrollRectToVisible:
              NSMakeRect(0, 0, 1, 1)];
```

The search action writes the search arguments into the user defaults database if the query was successful.

```
        [self writeArgs];
    }

    return self;
}
```

A browse window has a method for deallocation, which frees the space occupied by the directory string. Note that the memory from the buttons and textviews should be reclaimed automatically upon removal from the view hierarchy or window closure, since they were placed in an autorelease pool just after being created.

```
- (void)dealloc
{
    [directory release];
    [super dealloc];
}

@end
```

The main function has the usual format. It instantiates an autorelease pool, the controller and the application object.

```
int main(int argc, char** argv, char **env)
{
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    Controller *con = [Controller new];
    NSApplication *app = [NSApplication sharedApplication];
```

The application's menu includes an entry that lets the user open a directory with a browse window for searching. There is an entry for the action `copy:`, so that the user may copy text from the section textviews or the file textview into another application. The item "Quit" as usual quits the program.

```
    NSMenu *menu = [NSMenu new];
    [menu addItemWithTitle: @"Open"
         action:@selector(open:)
         keyEquivalent:@""];
    [menu addItemWithTitle: @"Copy"
         action:@selector(copy:)
         keyEquivalent:@""];
    [menu addItemWithTitle: @"Quit"
         action:@selector(terminate:)
         keyEquivalent:@"q"];
    [NSApp setMainMenu:menu];
```

It remains to set the application's delegate and run the application. Hopefully this little program illustrates successfully how a productivity tool may be implemented in a program of a few hundred lines.

```
    [app setDelegate:con];
    [app run];

    [pool release];
    exit(0);
}
```

Making the number of spaces that replace a tab customizable is left as an exercise.

# 4 Distributed objects

## 4.1 Connect four

*by Marko Riedel*

### 4.1.1 Idea

The idea behind this recipe is very simple: implement an application that allows two human players to play "connect four" over the network. The program displays the board to each player, who may drop chips of his own color in any one of the seven columns. The player who is the first to obtain four chips along a horizontal, vertical, or diagonal line, wins the game. The application detects a win and uses an alert panel to inform the players that the game is over. It also detects ties. The application has three states: "my turn", "opponent's turn" and "game over."

This game provides an opportunity to illustrate the use of distributed objects (DO) in an easy-to-understand application. There are two classes: a controller that interacts with the application object through delegation and a subclass of `NSView` that implements the board and the "connect four" functionality. There is one instance of the board, which the controller vends to the network so that the opponent may invoke methods to restart or quit the game, and record a move, all of it through a proxy. These methods are grouped in the protocol "connect four." The protocol works both ways: each player has a proxy of his opponent's board and uses it to communicate moves and restart or quit events to the process of his opponent. The connection is bi-directional rather than client-server. The name chosen by the controller to vend the board object has the form "connectfour-user-color", where "user" is the name of the user playing the game and "color" is the color the user chose (red or yellow, red begins). Start the game from the command line like this:

```
openapp DOConnectFour.app/ red 172.16.1.150 riedel.
```

### 4.1.2 Implementation

Start by including all the necessary headers and define the width and height of a single square on the board. Define an enumerated datatype that represents the contents of the board: a square is either empty or it contains a red or a yellow chip.

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

#define DIMENSION 50

typedef enum {
    FIELD_BLANK=0,
    FIELD_YELLOW,
    FIELD_RED
} FIELD_STATE;
```

The next set of constants defines the number of rows and columns, the total number of squares on the board and the dimensions of the board in pixels.

```
#define BOARD_WIDTH 7
#define BOARD_HEIGHT 6
#define BOARD_SIZE (BOARD_WIDTH*BOARD_HEIGHT)

#define TOTAL_WIDTH (BOARD_WIDTH*DIMENSION)
#define TOTAL_HEIGHT (BOARD_HEIGHT*DIMENSION)
```

The game has three states: "my turn", "opponent's turn" and "game over."

```
typedef enum {
   STATE_LOCAL_MOVE,
   STATE_REMOTE_MOVE,
   STATE_DONE
} STATE_GAME;
```

The protocol `ConnectFour` is of key importance. It defines the methods that the application may invoke on the proxy object representing the opponent's board. There are three methods: the first communicates to the opponent process to restart the game, the second, to quit the game and the third, to update the board and the state of the opponent's game to reflect a move by the local player. The qualifier `oneway void` indicates that the application needn't wait for the invoked method to return.

```
@protocol ConnectFour
- (oneway void)remoteNewGame;
- (oneway void)remoteQuit;
- (oneway void)remoteMoveAtRow:(int)row Col:(int)col Win:(BOOL)win;
@end
```

We may now define the class that represents the board. Note the first line – it says that it implements the protocol `ConnectFour`. The board stores the state of the game in the array `data`, which is `BOARD_WIDTH` columns wide and `BOARD_HEIGHT` rows high. The board records the number of moves and the state of the game, as well as the color of the local player (red or yellow). Finally there is an instance variable that holds the proxy object representing the opponent's board, which also implements the protocol `ConnectFour` (obviously, since it is an instance of the same class).

```
@interface Board : NSView <ConnectFour>
{
   FIELD_STATE data[BOARD_HEIGHT][BOARD_WIDTH];

   int moves;
   STATE_GAME state;
   FIELD_STATE color;

   id <ConnectFour> remote;
}
```

The class `Board` only contains a few simple methods. There is an initializer and a method to restart the game. There is an accessor to set the proxy object. The method `winner` determines whether the local player has a winning configuration. The most important methods are `drawRect:`, which draws the board and `mouseDown:`, which responds to a click on a column, advances the state of the game and notifies the opponent of the change.

```
- initWithColor:(FIELD_STATE)col;
- newGame;
- setRemote:(id <ConnectFour>)rem;

- (BOOL)winner;
```

```
- (void)drawRect:(NSRect)aRect;

- (void)mouseDown:(NSEvent *)theEvent;

@end
```

Now start the implementation. The initializer is very simple. It computes the frame rectangle and invokes NSView's method initWithFrame:. It records the color of the local player and restarts the game.

```
@implementation Board

- initWithColor:(FIELD_STATE)col
{
   NSRect frame;

   frame.origin = NSZeroPoint;
   frame.size.width = TOTAL_WIDTH;
   frame.size.height = TOTAL_HEIGHT;

   [super initWithFrame:frame];
   color = col;

   [self newGame];

   return self;
}
```

The method newGame is straightforward. It iterates over the squares of the board and resets every square to be empty.

```
- newGame
{
   int row, col;

   for(row=0; row<BOARD_HEIGHT; row++){
      for(col=0; col<BOARD_WIDTH; col++){
         data[row][col] = FIELD_BLANK;
      }
   }
```

It also resets the move counter to zero and determines the initial state based on the convention that red begins. The last step is to mark the view as needing redisplay.

```
   moves = 0;
   state =
      (color == FIELD_RED ?
       STATE_LOCAL_MOVE : STATE_REMOTE_MOVE);

   [self setNeedsDisplay:YES];

   return self;
}
```

The setter method setRemote: stores the proxy for the remote board in the corresponding instance variable. Retain and release for this object are handled by the controller.

```
- setRemote:(id <ConnectFour>)rem
{
```

```
    remote = rem;
    return self;
}
```

The method that detects winning configurations iterates over every square of the board. It checks whether the current square could be the first of a horizontal, vertical, ascending diagonal or descending diagonal segment of length four, and if it is, the segment's contents (chips) are checked. We have a winner if all four are the color of the local player.

```
- (BOOL)winner
{
    int row, col;

    for(row=0; row<BOARD_HEIGHT; row++){
        for(col=0; col<BOARD_WIDTH; col++){
```

Check horizontal segments first.

```
            if(col+3<BOARD_WIDTH){
                if(data[row][col] == color &&
                  data[row][col+1] == color &&
                  data[row][col+2] == color &&
                  data[row][col+3] == color){
                    return YES;
                }
            }
```

Do vertical segments next.

```
            if(row+3<BOARD_HEIGHT){
                if(data[row][col] == color &&
                  data[row+1][col] == color &&
                  data[row+2][col] == color &&
                  data[row+3][col] == color){
                    return YES;
                }
            }
```

Now try ascending diagonals.

```
            if(row+3<BOARD_HEIGHT && col+3<BOARD_WIDTH){
                if(data[row][col] == color &&
                  data[row+1][col+1] == color &&
                  data[row+2][col+2] == color &&
                  data[row+3][col+3] == color){
                    return YES;
                }
            }
```

Check descending diagonals last and return `NO` if no segment of length four was found.

```
            if(row-3>=0 && col+3<BOARD_WIDTH){
                if(data[row][col] == color &&
                  data[row-1][col+1] == color &&
                  data[row-2][col+2] == color &&
                  data[row-3][col+3] == color){
                    return YES;
                }
            }
```

```
        }
    }

    return NO;
}
```

The method `drawRect:` is important, yet straightforward. Start by allocating an array of colors for indexing with the `FIELD_STATE` data type.

```
- (void)drawRect:(NSRect)aRect
{
    int index, row, col;
    NSColor *cols[3] = {
        [NSColor blackColor],
        [NSColor yellowColor],
        [NSColor redColor]
    };
```

Set the line width to 1.0 and paint the background of the view blue.

```
    PSsetlinewidth(1.0);

    [[NSColor blueColor] set];
    PSrectfill(0, 0, TOTAL_WIDTH, TOTAL_HEIGHT);
```

The grid that delinates the squares is next. It is drawn in black. Left, right, upper and lower margins are not drawn. There are two loops: the first draws the vertical lines of the grid and the second one draws the horizontal ones.

```
    [[NSColor blackColor] set];
    for(index=1; index<BOARD_WIDTH; index++){
        PSmoveto(index*DIMENSION, 0);
        PSlineto(index*DIMENSION, TOTAL_HEIGHT);
    }
    for(index=1; index<BOARD_HEIGHT; index++){
        PSmoveto(0, index*DIMENSION);
        PSlineto(TOTAL_WIDTH, index*DIMENSION);
    }
    PSstroke();
```

The last step is to draw the chips, or a black filled circle for empty squares. We iterate over the data array, compute the center of the filled arc and use the data value as an index into the array of colors. We set the color and fill the arc.

```
    for(row=0; row<BOARD_HEIGHT; row++){
        for(col=0; col<BOARD_WIDTH; col++){
            NSPoint pt = {
                col*DIMENSION+DIMENSION/2,
                row*DIMENSION+DIMENSION/2
            };
            [cols[data[row][col]] set];
            PSarc(pt.x, pt.y, DIMENSION/3, 0, 360);
            PSfill();
        }
    }
}
```

The way the board responds to clicks by the user is determined by the method `mouseDown:`. If the state of the game is not equal to "my turn" i.e. `STATE_LOCAL_MOVE`, then the click is not valid, and the user hears a beep.

```
- (void)mouseDown:(NSEvent *)theEvent
{
    if(state!=STATE_LOCAL_MOVE){
        NSBeep();
        return;
    }
```

We need to know where the user has clicked and retrieve the corrdinates where the mouse-down occurred. We convert these coordinates to view coordinates. We have a case where the view is the content view of the window, so the conversion is not strictly necessary, but it is the right thing to do, since a view may be anywhere in a window.

```
    NSPoint curp;
    curp = [theEvent locationInWindow];
    curp = [self convertPoint:curp fromView: nil];
```

Next we compute the column where the user clicked. We use a loop to start at the top of the column and look for the highest square that is not empty. We break out of the loop if we find one.

```
    int row, col;
    col = curp.x/DIMENSION;
    for(row=BOARD_HEIGHT-1; row>=0; row--){
        if(data[row][col]!=FIELD_BLANK){
            break;
        }
    }
```

If the highest square that is not empty is the top square of the column, then we know that the user clicked on a column that is full, and the application beeps to inform the user of this fact.

```
    if(row==BOARD_HEIGHT-1){
        NSBeep();
    }
```

The remaining case is the case of a valid move. We record the move in the data array and display the view.

```
    else{
        moves++; row++;
        data[row][col] = color;

        [self display];
```

It remains to check whether the move resulted in a winning configuration. We check and record the result in `win`, and communicate the state change to the other player.

```
        BOOL win = [self winner];

        [remote
            remoteMoveAtRow:row Col:col Win:win];
```

The game goes into the "game over" state in the case of a winning move. We pop up an alert panel to congratulate the user.

```
        if(win){
            state = STATE_DONE;
            NSString *msg =
                [NSString stringWithFormat:@"%@ wins!",
                        (color==FIELD_RED ? @"Red" : @"Yellow")];
```

```
        NSRunAlertPanel(@"Congratulations!", msg,
                        @"Ok", nil, nil);
    }
```

The game also goes into the "game over" state if there are no more free squares available, in which case we have a tie, and alert the user.

```
    else if(moves==BOARD_SIZE){
        state = STATE_DONE;
        NSRunAlertPanel(@"Game over.", @"Tie (red, yellow).",
                        @"Ok", nil, nil);
    }
```

If we didn't win and there is room on the board, then it must be the opponent's turn, and we make the appropriate state change.

```
    else{
        state = STATE_REMOTE_MOVE;
    }
  }
}
```

Note that we recorded the state change before we displayed the alert panel in the two cases where an alert was necessary. This is because the alert panel runs a modal loop during which our board could receive events from the opponent. The way the code is written we will not overwrite any state changes that could occur while the alert panel is on screen.

The last section of the board class implements the protocol `ConnectFour`, which we saw earlier, and which makes communication between the two players possible. The first method (`remoteNewGame`) is invoked when the opponent decides to restart the game. We do the same, thus keeping the two boards in sync.

```
- (oneway void)remoteNewGame
{
  [self newGame];
}
```

The second method is invoked when the opponent quits the game. We are observing the notification "connection died", which we stop observing when the opponent quits. We terminate the application once this is done.

```
- (oneway void)remoteQuit
{
  [[NSNotificationCenter defaultCenter]
     removeObserver:self];
  [[NSApplication sharedApplication] terminate:self];
}
```

The method remoteMoveAtRow:Col:Win is the most important one of the protocol and it keeps the two boards in sync. It starts by incrementing the move counter, records the move in the data array and displays the board.

```
- (oneway void)remoteMoveAtRow:(int)row Col:(int)col Win:(BOOL)win
{
  moves++;
  data[row][col] =
     (color==FIELD_RED ? FIELD_YELLOW : FIELD_RED);

  [self display];
```

If the opponent reached a winning position, then we lost the game, change state to "game over" and run an appropriate alert.

```
    if(win){
        state = STATE_DONE;
        NSString *msg =
            [NSString stringWithFormat:@"%@ loses.",
                    (color==FIELD_RED ? @"Red" : @"Yellow")];
        NSRunAlertPanel(@"Game over.", msg,
                    @"Ok", nil, nil);
    }
```

We also go into the state "game over" if there is no more room on the board. This is exactly the same as in the method `mouseDown:`, and we also run an alert panel.

```
    else if(moves==BOARD_SIZE){
        state = STATE_DONE;
        NSRunAlertPanel(@"Game over.", @"Tie (red, yellow).",
                    @"Ok", nil, nil);
    }
```

The last case occurs if the opponent's move wasn't a winning one and there is room on the board. In that case it's our turn; by the way, the observation about not changing state after the panel appears applies here as well.

```
    else{
        state = STATE_LOCAL_MOVE;
    }
}

@end
```

The last part of this recipe is the controller. It responds to two notifications: "application finished launching" and "connection died." It stores the window and the board in instance variables, as well as two strings that describe the two players and a proxy object that represents the board object of the opponent.

```
@interface Controller : NSObject
{
    NSWindow *window;
    Board *board;

    NSString *localName, *remoteName;
    id <ConnectFour> remote;
}
```

The first two methods respond two the two notifications described above. The remaining two methods are action methods that respond to clicks on the main menu. They restart the game and terminate the application, respectively.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif;
- (void)connectionDied: (NSNotification *)notif;

- newGame:(id)sender;
- terminate:(id)sender;
@end
```

It is the job of `applicationDidFinishLaunching:` to build the two components of the GUI: the main menu and the window that displays the board. It must also read command line parameters that determine the color of the local player and the host and username of the opponent. It first retrieves the array of arguments and checks that we have the right number and raises an exception otherwise.

```
@implementation Controller

- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   NSProcessInfo *procInfo = [NSProcessInfo processInfo];
   NSArray *args = [procInfo arguments];

   if([args count]!=4){
      [NSException raise:NSInvalidArgumentException
                  format:@"args are: <color> <host> <user>"];
   }
```

The first argument gives the color of the local player. We retrieve it from the argument array and compare it against the strings "red" and "yellow" and store the result in the variable `lcolor`. We raise an exception if the user entered an unknown color.

```
   NSString *colName = [args objectAtIndex:1];
   FIELD_STATE lcolor;
   if([colName isEqualToString:@"red"]){
      lcolor = FIELD_RED;
   }
   else if([colName isEqualToString:@"yellow"]){
      lcolor = FIELD_YELLOW;
   }
   else{
      [NSException raise:NSInvalidArgumentException
                  format:@"color must be red or yellow"];
   }
```

The three parameters `remoteHost`, `remoteUser` and `remoteColor` describe the opponent. They are initialized from the array of arguments.

```
   NSString
      *remoteHost = [args objectAtIndex:2],
      *remoteUser = [args objectAtIndex:3],
      *remoteColor =
      (lcolor == FIELD_RED ?
       @"yellow" : @"red");
```

The next step is to allocate and initialize the board and the window whose content view it will become. The window will not be resizable.

```
   board = [[Board alloc] initWithColor:lcolor];

   window =
      [[NSWindow alloc]
         initWithContentRect:[board frame]
         styleMask:NSTitledWindowMask
         backing:NSBackingStoreRetained
         defer:NO];
```

The title of the window shows the color of the local player and the name and host of the opponent. We make the board the content view of the window and set the window delegate.

```
   NSString *title =
      [NSString
         stringWithFormat:@"Connect Four: %@ vs. %@ - %@",
         colName, remoteHost, remoteUser];
```

```
[window setTitle:title];

[window setContentView:board];
[window setDelegate:self];
```

The variable `localName` holds the name under which we will register the local connection and make the local board available to the opponent.

```
localName =
   [NSString
      stringWithFormat:
         @"connectfour−%@−%@", NSUserName(), colName];
[localName retain];
```

We retrieve the default connection, set its root object and attempt to register it under the local name. We raise an exception if we couldn't register under the local name.

```
NSConnection *conn = [NSConnection defaultConnection];

[conn setRootObject:board];
if(![conn registerName:localName]){
   [NSException raise:NSGenericException
            format:@"couldn't␣register␣%@", localName];
}
```

The next step is to constuct the name of the remote connection. It has the same form as the local name. (This is important.) It contains the remote user and color and a prefix that identifies the application.

```
remoteName =
   [NSString
      stringWithFormat:
         @"connectfour−%@−%@", remoteUser, remoteColor];
[remoteName retain];
```

We now try to establish a connection. We try this sixty times, with a pause of three seconds between each attempt, for a total of three minutes. We try to obtain the proxy object under the remote name at the remote host. We sleep for three seconds if we fail and then we try again. We log each attempt so that the user sees what the application is doing.

```
#define MAXTRIES 60
#define SLEEPSECS 3
int tries = 0;
do {
   remote = (id <ConnectFour>)
      [NSConnection
         rootProxyForConnectionWithRegisteredName:
            remoteName host:remoteHost];
   if(remote==nil){
      [NSThread
         sleepUntilDate:
            [NSDate dateWithTimeIntervalSinceNow:
                     SLEEPSECS]];
   }
   NSLog(@"trying␣%@␣%@", remoteHost, remoteName);
} while(remote==nil && tries++ < MAXTRIES);
```

If the variable `remote` is nil after the loop has finished, then we failed to obtain a connection and raise an exception. Otherwise we retain the proxy object and set the corresponding instance variable of the local board object.

```
   if(remote==nil){
      [NSException raise:NSGenericException
               format:@"couldn't connect to %@", remoteName];
   }
   [(NSObject *)remote retain];
   [board setRemote:remote];
```

We add ourselves to the default notification center as an observer of the notification "connection died." We want to be notified so that we may terminate the application.

```
   [[NSNotificationCenter defaultCenter]
      addObserver:self
      selector:@selector(connectionDied:)
      name:NSConnectionDidDieNotification
      object:[(NSDistantObject *)remote connectionForProxy]];
```

We may order the window to the front once we have registered under the local name and obtained the proxy object of our opponent. The window is centered on screen.

```
   [window center];
   [window orderFrontRegardless];
   [window makeKeyWindow];
```

The last step is to construct the main menu, which has two entries, one to restart the game and another one to quit the application. Now the application is ready to go.

```
   NSMenu *menu = [NSMenu new];

   [menu addItemWithTitle: @"New Game"
        action:@selector(newGame:)
        keyEquivalent:@"n"];
   [menu addItemWithTitle: @"Quit"
        action:@selector(terminate:)
        keyEquivalent:@"q"];
   [NSApp setMainMenu:menu];

   [menu display];
}
```

The next method responds to the notification that the connection died. We don't do anything fancy here; we release the proxy object and raise an exception.

```
- (void)connectionDied: (NSNotification *)notif
{
   [(NSObject *)remote release];
   [NSException raise:NSGenericException
            format:@"connection died: %@", remoteName];
}
```

The two action methods are last. The method `newGame:` is invoked from the menu. It communicates the state change to the opponent and resets the local state and board.

```
- newGame:(id)sender
{
   [remote remoteNewGame];
   [board newGame];
   return self;
}
```

```
Directory listing of /home/gnustep/gswebsrv/

Up one level.

 -  GNUmakefile                2003-07-08 16:53:23 +0200              159
 -  GNUmakefile~               2003-07-08 16:52:40 +0200              156
 -  gswebsrv.m                 2003-07-09 18:39:03 +0200             6565
 -  gswebsrv.m~                2003-07-09 12:52:37 +0200             2417
[+] obj                        2003-07-08 17:58:17 +0200               39
[+] shared_obj                 2003-07-08 17:58:17 +0200             4096
```

Figure 6: Example web server session with `w3m`.

The last method in this recipe is the method `terminate:`, which is also invoked from the menu. We do not want to receive the notification "connection died" when the opponent's application quits in response to our message. Hence we remove ourselves from the notification center. The next step is to tell the other side to quit. We terminate the application once this is done.

```
- terminate:(id)sender
{
   [[NSNotificationCenter defaultCenter]
      removeObserver:self];
   [remote remoteQuit];
   [[NSApplication sharedApplication] terminate:self];
}

@end
```

This is the end of the recipe, which is based on the DO tutorials by Nicola Pero and Adam Fedor. We do not show the function `main`, since it is identical to what we have e.g. in the n-queens recipe.

# 5 CGI programming

## 5.1 Web server

*by Marko Riedel*

### 5.1.1 Idea

The goal is to implement a web server that lets the client navigate the server's file system. It should respond in two different ways. It should send the contents of the file if a file is requested. If a directory is requested, it should send a directory listing to the client so that the user can navigate the file system. Figure 6 shows the server display a page with `w3m`.

### 5.1.2 Preliminaries

We'll be using `xinetd` to handle requests to the web server. This way we don't have to worry about implementing a complete daemon that forks a child process for every request etc., `xinetd` will handle all of this for us. This is the configuration that we use.

```
service gswebsrv
{
        port            = 7788
        socket_type     = stream
        wait            = no
        user            = wwwrun
```

```
        group           = nogroup
        server          = /usr/sbin/gswebsrv.sh
}
```

It is basically self-explanatory. We specify user and group for the server as well as the port. The entry "server" points to a shell script (`bash`) that performs some initialization. It sets the requisite GNUstep variables, the user and the time zone and invokes the actual server that will handle the request. It is important to set these variables correctly, or else warnings by GNUstep will be output to the web server. Needless to say the home directory of `wwwrun` has to exist for this to work properly.

```
#! /bin/sh
#

export LOGNAME=wwwrun
export HOME=/home/wwwrun
export TZ=Europe/Berlin

GNUSTEP_SYSTEM_ROOT=/usr/GNUstep
export GNUSTEP_SYSTEM_ROOT
. $GNUSTEP_SYSTEM_ROOT/System/Makefiles/GNUstep.sh

/home/gnustep/gswebsrv/shared_obj/ix86/linux-gnu/gnu-gnu-gnu/gswebsrv
```

Given this setup, `xinetd` will listen on port `7788`, fork a process for every request, connect the socket to the standard input and output of the process and invoke the shell script, which in turn sets the necessary variables and starts the actual server. The server only has to read the request from the standard input and produce a response on the standard output.

### 5.1.3 Implementation

HTTP knows many different status codes, of which we'll be using quite a few, mostly to signal errors. The server sends a request followed by a set of headers, which we'll ignore in our server. The client sends a status code, followed by a set of headers and the response data.

These are the codes that our server will produce:

- 220 OK
- 400 Bad Request
- 403 Forbidden
- 404 Not Found
- 501 Not implemented
- 505 HTTP Version Not Supported

We could have done with fewer codes, but we use them here to illustrate the problem of writing a web server. All these codes with the exception of the first one are error codes, so we start with a routine that handles errors. It outputs the error code followed by a detailed description of the problem, and exits the server (recall that there is one server process for every request). The routine constructs the code description from the status code itself and the description and stores it in the string `msg`. These are the entries shown in the list above. Subsequently the routine builds the body of the response, whose title contains the code description. The body lists the description and any explanatory details we might have supplied. It includes the process name and the host name so that both can be identified easily during debugging. The last step is to output the status code, the content type of the message body, the HTML body itself and exit the server.

```
#include <Foundation/Foundation.h>
#include <fcntl.h>
```

```
void ErrmsgAndExit(int code, NSString *desc, NSString *detail)
{
   NSProcessInfo *pinfo = [NSProcessInfo processInfo];

   NSString *msg =
      [NSString stringWithFormat:@"%d %@", code, desc];
   NSString *body =
      [NSString stringWithFormat:@"<HTML><HEAD>\n"
              @"<TITLE>%@</TITLE>\n"
              @"</HEAD><BODY BGCOLOR=white>\n"
              @"<H1>%@</H1>\n"
              @"%@\n"
              @"<HR>\n"
              @"<ADDRESS>%@ at %@</ADDRESS>\n"
              @"</BODY></HTML>\n",
              msg, desc, detail,
              [pinfo processName], [pinfo hostName]];


   printf("HTTP/1.1 %s\r\n", [msg cString]);
   printf("Content-type: text/html\r\n\n");
   printf("%s", [body cString]);

   exit(1);
}
```

The routine `main` must process the two types of requests. The very first thing it needs to do is to read the request from the standard input. For this purpose we obtain the appropriate file handle and read any data that might be available. We need to know how many bytes have been read, so that we can convert the contents of the data object into a string.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   NSData *reqData =
      [[NSFileHandle fileHandleWithStandardInput]
         availableData];

   unsigned reqLength = [reqData length];
   const char *reqBytes = [reqData bytes];
```

We convert the data to a string and split the request into lines. We are only interested in the first line (the request itself), but we could conceivably expand the server to do additional processing on the headers that are stored in the array `lines`.

```
   NSArray *lines;
   NSString *firstLine;

   lines = [[NSString stringWithCString:reqBytes length:reqLength]
            componentsSeparatedByString:@"\n"];
   firstLine = [lines objectAtIndex:0];
```

A good request line should look like this:

```
GET /path/to/file HTTP/1.1
```

Hence we need to split it into fields so that we can check the components of the request and deliver the result. We use code that is also present in the `df` recipe, i.e. we obtain a character set that contains white

space and a scanner whose data source is the first line we have read. We scan fields until the scanner reaches the end of the line.

```
NSCharacterSet *space =
   [NSCharacterSet whitespaceAndNewlineCharacterSet];
NSScanner *scn =
   [NSScanner scannerWithString:firstLine];
NSMutableArray *fields =
   [NSMutableArray arrayWithCapacity:3];
NSString *field;

while([scn isAtEnd]==NO){
   [scn scanCharactersFromSet:space
       intoString:NULL];
   [scn scanUpToCharactersFromSet:space
       intoString:&field];
   [fields addObject:field];
}
```

A series of validity checks follows. There must be three fields: the request, the entity requested and the version of the protocol.

```
if([fields count]!=3){
   ErrmsgAndExit(400, @"Bad Request",
             @"Format is: GET <path> <version>.");
}
```

Our server only responds to `GET` requests, upon receipt of which it produces the contents of the file or a directory browser.

```
if([[fields objectAtIndex:0] isEqualToString:@"GET"]==NO){
   ErrmsgAndExit(501, @"Not Implemented",
             @"Use GET only.");
}
```

We will be picky about the protocol that we support, it must be either HTTP 1.0 or HTTP 1.1.

```
field = [fields objectAtIndex:2];
if([field isEqualToString:@"HTTP/1.0"]==NO &&
  [field isEqualToString:@"HTTP/1.1"]==NO){
   ErrmsgAndExit(505, @"HTTP Version Not Supported",
             @"Use HTTP/1.0 or HTTP/1.1.");
}
```

This ends the series of request checks. We now know that we have a good request. However, it is entirely possible that the client has requested a file or a directory that does not exist. We get the default file manager and ask it to check the path from the request. We return "Not Found" if there is no file or directory at that path.

```
NSString *path = [fields objectAtIndex:1];
NSFileManager *fm = [NSFileManager defaultManager];
BOOL isDir;

if([fm fileExistsAtPath:path isDirectory:&isDir]==NO){
   NSString *detail =
      [NSString stringWithFormat:@"The requested URL "
             @"%@ was not found on this server.", path];
   ErrmsgAndExit(404, @"Not Found", detail);
}
```

The last check is to determine whether we can read the file or directory.

The method `isReadableFileAtPath` uses the `access(2)` call and works on files and directories. We send an error message if we cannot access the requested entity.

```
// "access" call works on files and directories
if([fm isReadableFileAtPath:path]==NO){
   NSString *detail =
      [NSString stringWithFormat:@"You don't have "
              @"permission to accesss %@ "
              @"on this server.", path];
   ErrmsgAndExit(403, @"Forbidden", detail);
}
```

We can send a `220 OK` response now that the request and the requested entity have been verified.

```
printf("HTTP/1.1 220 OK\r\n");
```

Start with the easy part, i.e. serving files. We use a minimal set of headers, namely the content type and the content length. The content type is `application/octet-stream` and the content length is obtained from the file manager.

```
if(isDir==NO){
   NSDictionary *attrib =
      [fm fileAttributesAtPath:path traverseLink:YES];

   printf("Content-type: application/octet-stream\r\n");
   printf("Content-length: %d\r\n\n",
        [[attrib objectForKey:NSFileSize] intValue]);
   fflush(stdout);
```

It remains to send the contents of the file. We could read the entire file into a data object and send the file all at once, but this could result in a server process requesting a lot of memory. Instead we choose to serve the file in chunks of 64K.

We obtain a file handle for reading the file and the file handle for standard output (an effort was made to use foundation objects rather than system calls). We read one chunk after another and output the current chunk immediately. We close the file when there are no more data.

```
   #define CHUNK_SIZE (1<<16)
   NSFileHandle *reader =
      [NSFileHandle fileHandleForReadingAtPath:path];
   NSFileHandle *writer =
      [NSFileHandle fileHandleWithStandardOutput];
   NSData *chunk;

   while(chunk=[reader readDataOfLength:CHUNK_SIZE],
        [chunk length]>0){
      [writer writeData:chunk];
   }

   [reader closeFile];
}
```

That's it for the case of serving files. Producing a browsable directory listing requires a bit more effort. We start by ensuring that the path ends in a path delimiter and append one if this is not the case.

```
else{
   if([path hasSuffix:@"/"]==NO){
      path = [path stringByAppendingString:@"/"];
   }
```

We start by constructing the title of the HTML document that we will serve, and output the content type and the beginning of the document (title, open body tag, background color).

```
NSString *title =
   [NSString stringWithFormat:@"Directory listing "
          @"of %@", path];

printf("Content-type: text/html\r\n\n");

printf("<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n",
     [title cString]);
printf("<BODY BGCOLOR=white>\n");
```

We wish to have a certain feature to simplify navigation. There should be a header that displays the current directory in such a manner that directories that are higher up in the tree are clickable, e.g. if the path is `/path/to/subdirectory/`, then both `path` and `to` should be clickable and take the user to `/path` and `/path/to`, respectively.

The first step is to split the path into components.

```
NSArray *compArray = [path pathComponents];
int cind, cmax = [compArray count];
printf("<H1>Directory listing of /");
```

A path like `/path/to/subdirectory/` yields five components, the first and last of which are slashes; the root directory yields a single component. We process the inner components of the path, e.g. `path`, `to` and `subdirectory`. We construct the subpaths for each inner component excluding the last one, giving `/path` and `/path/to`. We output an anchor for each component. The anchor points to the complete path and contains the last component of the subpath for display. The last component of the complete path is not displayed in this manner because it points to the directory being displayed (that's why we have `cind<cmax-2` rather than `cind<cmax-1`.)

```
for(cind=1; cind<cmax-2; cind++){
   NSString *subPath =
      [[compArray subarrayWithRange:NSMakeRange(1, cind)]
         componentsJoinedByString:@"/"];
   printf("<A HREF=/%s>%s/</A>",
        [subPath cString],
        [[compArray objectAtIndex:cind] cString]);
}
```

It remains to display the last component, which is not clickable. There is no last component when we browse the root directory. This concludes the construction of the navigable header.

```
if(cmax>1){
  printf("%s/", [[compArray objectAtIndex:cind] cString]);
}
printf("</H1>\n");
```

The user must be able to ascend the directory tree after he has descended it in search of some file or directory. We output an anchor for this purpose. If we are not browsing the root directory, then output an anchor with the title `Up one level`, which points to the parent directory.

```
if([path isEqualToString:@"/"]==NO){
   printf("<A HREF=%s>Up one level.</A><P>\n",
        [[path stringByDeletingLastPathComponent]
           cString]);
}
```

We are now ready to enumerate the contents of the directory. Start by obtaining the contents of the directory (which do not include ‘‘.’’ and ‘‘..’’, by the way). Sort them alphabetically, but ignoring case, and fetch the enumerator of the sorted array. The string `item` will hold a single entry.

```
    NSEnumerator *dirContentEnum =
       [[[fm directoryContentsAtPath:path]
           sortedArrayUsingSelector:
              @selector(caseInsensitiveCompare:)]
          objectEnumerator];
    NSString *item;
```

We produce one line of output for each item and iterate over the items with the enumerator. We construct the full path to each item and check whether it is a directory or not.

```
    printf("<PRE>\n");
    while((item = [dirContentEnum nextObject])!=nil){
       NSString *fullPath =
          [path stringByAppendingString:item];

       BOOL itemIsDir;
       [fm fileExistsAtPath:fullPath isDirectory:&itemIsDir];
```

The anchor for the item points to the full path and lists the item, i.e. the last component of the full path. We mark directories with the string ``[+]'', chosen because it resembles the icon that is used for directories by some graphical browsers.

```
       NSString
          *fmt = @"<A␣HREF=%@>%@␣%@</A>",
          *anchor =
          [NSString stringWithFormat:fmt, fullPath,
                 (itemIsDir==YES ? @"[+]" : @"␣–␣"),
                 item];
       printf("%s", [anchor cString]);
```

We pad the line up to column sixty, so that the modification time and the size of the file line up properly when we output them, which we'll do next.

```
       int ilen = 3+1+[item length];
       while(ilen++<60){
          putchar('␣');
       };
```

We obtain the attributes of the current item from the file manager and extract the modification date and the size. By the way, this is where the variable `TZ` from the shell script comes into play. We output the date and the file size. This ends the current iteration of the loop. Note that the entire listing will be displayed as-is, because it is bracketed by `PRE` tags.

```
       NSDictionary *itemAttrib =
          [fm fileAttributesAtPath:fullPath traverseLink:NO];
       NSString *dateStr =
          [[itemAttrib fileModificationDate] description];
       printf("%s␣%16d\n", [dateStr cString],
            [[itemAttrib objectForKey:NSFileSize] intValue]);
    }
    printf("</PRE>\n");
```

The last step is to close the `BODY` and `HTML` tags, flush buffers, and exit the server. Easy, wasn't it?

```
    printf("</BODY></HTML>\n");

    fflush(stdout);
}
```

```
   [pool release];
   exit(0);
}
```

## 5.2 Read GET and POST variables from forms and URLs

*by Marko Riedel*

### 5.2.1 Idea

We add a category to `NSMutableDictionary` that contains a method that initializes the dictionary with the variables that were passed to the CGI script.

The procedure is simple.

1. Obtain the data from environment variables: if `REQUEST_METHOD` is `POST` read `CONTENT_LENGTH` bytes from the standard input; if it is `GET`, use the contents of `QUERY_STRING`.

2. Split the data into key-value pairs. This is done by splitting on the ampersand character `&`. The data consists of a sequence of such pairs, where the equal sign `=` denotes assignment, as in

   `key1=value1&key2=value2.`

3. Replace the plus character `+` by a space and hexadecimal escape sequences of the form `%XX` by the corresponding ASCII character in both the key and the value.

### 5.2.2 Implementation

We start by adding a method to `NSString` that will carry out the last of the steps that we described above. It will scan the respective `C` string and and write characters into a temporary buffer. We allocate the same number of bytes as in the original string since it can only get shorter.

```
@implementation NSString (CGI)

- parseCGI
{
   int pos, length = [self length];
   const char *cstr = [self cString];

   NSZone *zone = NSDefaultMallocZone();

   char *rbuf, *rcur;
   NSString *result;

   rbuf = rcur =
      NSZoneMalloc(zone, length*sizeof(char));
```

We use a loop to do the actual processing. A plus character is recorded as a space character. A complete hexadecimal escape sequence is passed to `sscanf` for conversion into an unsigned integer, the lowest byte of which is recorded in the buffer.

```
   for(pos=0; pos<length; pos++){
      if(cstr[pos]=='+'){
         *rcur++ = ' ';
      }
      else if(cstr[pos]=='%' && pos+2<length &&
            isxdigit(cstr[pos+1]) && isxdigit(cstr[pos+2])){
         char hex[3] = { cstr[pos+1], cstr[pos+2], 0 };
         unsigned int dec;
```

```
            sscanf(hex, "%x", &dec);
            pos += 2;

            *rcur++ = dec;
        }
        else{
            *rcur++ = cstr[pos];
        }
    }
    result = [NSString stringWithCString:rbuf length:rcur-rbuf];
    NSZoneFree(zone, rbuf);

    return result;
}
```

Finally the method converts the data into the result string and frees the temporary buffer.

The method `initWithCGI` (added to `NSMutableString`) actutally creates the dictionary. We prepare for this by reading the process environment and immediately consulting `REQUEST_METHOD` to determine how to obtain the raw data. We will scan the data from start to finish and declare three variables to maintain state during the scan: `prev`, the first character of the current key-value pair, `eqpos`, the position of the equal sign in the current pair, and `pos`, which gives the first position of the next pair or the end of the data.

```
@implementation NSMutableDictionary (CGI)

- (id)initWithCGI
{
    NSProcessInfo *procInfo = [NSProcessInfo processInfo];
    NSDictionary *env = [procInfo environment];

    NSString *method = [env objectForKey:@"REQUEST_METHOD"];

    int length, prev, pos, eqpos;
    const char *bytes;
```

We obtain the data first. `REQUEST_METHOD` tells us how. We raise an exception if the data available on the standard input do not contain the number of bytes specified in `CONTENT_LENGTH` (this applies to `POST` submissions). We initialize the dictionary by preparing it to hold eight pairs (we could have chosen a different capacity).

```
    if([method isEqualToString:@"GET"]){
        NSString *str = [env objectForKey:@"QUERY_STRING"];

        length = [str length];
        bytes = [str cString];
    }
    else if([method isEqualToString:@"POST"]){
        NSData *data;

        length = [[env objectForKey:@"CONTENT_LENGTH"]
                  intValue];
        data = [[NSFileHandle fileHandleWithStandardInput]
                 readDataOfLength:length];
        if([data length]!=length){
            NSString *fmt =
                @"expected_%d_characters_on_STDIN,_got_%d";
            [NSException raise:NSRangeException format:fmt,
                       length, [data length]];
```

```
    }
    bytes = [data bytes];
}
else{
    [NSException raise:NSInvalidArgumentException
              format:@"request method not supported: %@",
              method];
}

[self initWithCapacity:8];
```

Now start the loop. We record the most recent occurence of the equal sign for use in the key-value split.

```
for(prev=0, pos=0; pos<length; pos++){
    if(bytes[pos]=='='){
        eqpos = pos;
    }
```

An ampersand signals that a complete pair has been scanned, as does having scanned up to the last character of the string. We compute the length of the key string and the value string and make sure they contain meaningful values, e.g. the key should not be empty.

```
    if(bytes[pos]=='&' || (pos>0 && pos==length-1)){
        int klen, vlen;
        NSString *key, *value;
        id lookup;

        klen = eqpos-prev;
        vlen = pos-eqpos-1+(bytes[pos]=='&' ? 0 : 1);

        if(!(klen>0 && vlen>=0)){
            [NSException raise:NSInvalidArgumentException
                      format:@"malformed query/content"];
        }
```

It remains to convert the source bytes into Objective C strings and parse them with `parseCGI` as discussed previously.

```
        key =
            [[NSString
                stringWithCString:bytes+prev
                length:klen] parseCGI];
        value =
            [[NSString
                stringWithCString:bytes+eqpos+1
                length:vlen] parseCGI];
```

Now put the pair into the dictionary. There are three cases.

1. The key is not in the dictionary. We add the key-value pair.
2. The key is already in the dictionary but it is not an array. We have found a second value for the key, and replace the object that it refers to by an array containing the two values.
3. The key is in the dictionary and refers to an array. We add the new value to the array.

Multiple values for the same key arise when the state of a set of checkboxes is submitted. We store the start position of the next pair in `prev` at the end of the body of the loop.

```
        lookup = [self objectForKey:key];

        if(lookup==nil){
           [self setObject:value forKey:key];
        }
        else if([lookup isKindOfClass:
                        [NSMutableArray class]]==NO){
           [self setObject:[NSMutableArray
                           arrayWithObjects:value,
                           lookup, nil]
              forKey:key];
        }
        else{
           [lookup addObject:value];
        }

        prev = pos+1;
     }
  }

  return self;
}
```

The routine returns `self`, which now contains all pairs that were submitted.

This code was inspired by `cgi-lib.pl` by Steven E. Brenner.

## 5.3 Working with cookies

*by Marko Riedel*

### 5.3.1 Idea

Cookies are a mechanism by which the `HTTP` protocol, which is stateless, can add persistence to a session. This is initiated by the server, which sends a cookie that the client stores locally. The client subsequently includes the cookie in its HTTP headers when it requests a page from the server (precisely what servers will receive a cookie is controlled by the path and domain properties of the cookie; cookies are sent with requests whose URL matches the path and the domain). The server could use cookies to store information that it has received from the client through a form submission. Usually just one cookie is used, however, namely the session id. The server maps the session id to the session state data.

We add a category to `NSMutableDictionary` that contains a method that initializes the dictionary with the variables that were passed in through the environment variable `HTTP_COOKIE`. We also implement a method that outputs the content of a dictionary as a series of HTTP `Set-Cookie` headers.

Our test application adds two cookies to the set of cookies it has received from the client. These cookies have a lifetime of thirty seconds. The value of each cookie is the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds, so that the server can output the remaining time for each cookie it receives.

The syntax of the `Set-Cookie` header is like this (all of it on one line):

```
Set-Cookie: KEY=VALUE
[; expires=DATE] [; path=PATH]
[; domain=DOMAIN_NAME] [; secure]
```

The date is required to have the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`.

The `Cookie` header that is returned by the client looks like this:

```
KEY1=VALUE1; KEY2=VALUE2; ...
```

Note that the protocol as we use it here lets the server receive multiple key-value pairs in a single `Cookie` header, whereas we must send one `Set-Cookie` header for each pair we wish to set.

### 5.3.2 Preliminaries

We use a shell script as a wrapper; it should go into the CGI-BIN directory of your webserver, e.g. you could save it as `/cgi-bin/cookies.sh`.

```
#! /bin/sh
#

export GNUSTEP_SYSTEM_ROOT=/usr/GNUstep
export TZ=Europe/Berlin
. $GNUSTEP_SYSTEM_ROOT/System/Makefiles/GNUstep.sh

/home/gnustep/cookies/shared_obj/ix86/linux-gnu/gnu-gnu-gnu/cookies
```

### 5.3.3 Implementation

Start by declaring `PERSIST`, a constant that defines the persistence of the cookies we produce. The method `initWithCookie` parses `HTTP_COOKIE` and stores the key-value pairs in the dictionary. The method `cookieToString` returns a string that is suitable for inclusion among the headers sent to the client. The special keys `expires, domain, path` and `secure` can be used to set the properties of the cookie.

```
#include <Foundation/Foundation.h>

#define PERSIST 30

@interface NSMutableDictionary (Cookies)

- (NSMutableDictionary *)initWithCookie;
- (NSString *)cookieToString;

@end
```

The first step is to retrieve the cookie from the process environment. There is work to be done if `HTTP_COOKIE` was not empty.

```
@implementation NSMutableDictionary (Cookies)

- (id)initWithCookie
{
   NSString *cookie =
      [[[NSProcessInfo processInfo] environment]
         objectForKey:@"HTTP_COOKIE"];

   [self initWithCapacity:1];

   if(cookie!=nil){
```

The algorithm works works like this: scan up to the first semicolon and split the key-value pair on the equal sign. Skip whatever whitespace you may find. Repeat until you reach the end of the string.

We need a character set for whitespace so that we may skip those characters. We intialize the scanner with the cookie and start a loop that ends when the scanner reaches the end of the string.

```
      NSCharacterSet *space =
         [NSCharacterSet whitespaceAndNewlineCharacterSet];
      NSScanner *scn =
         [NSScanner scannerWithString:cookie];

      while([scn isAtEnd]==NO){
```

We scan up to the first semicolon; `pair` holds those characters, but does not include the semicolon. We split the pair on the equal sign.

```
        NSString *pair;
        if([scn scanUpToString:@";" intoString:&pair]==YES){
           NSArray *entry =
              [pair componentsSeparatedByString:@"="];
           NSString *key, *value;
```

We do some error checking. There should be exactly two fields: a key and a value. We raise an exception if this is not the case.

```
           if([entry count]!=2){
              [NSException
                 raise:NSRangeException
                 format:@"cookie \"%@\""
                 @"contains bad key-value pair; "
                 @"%d fields, should be 2; ",
                 cookie, [entry count]];
           }
```

Neither the key nor the value may be empty.

```
           key = [entry objectAtIndex:0];
           value = [entry objectAtIndex:1];

           if(![key length] || ![value length]){
              [NSException
                 raise:NSRangeException
                 format:@"cookie \"%@\":"
                 @"key and value must not be empty;"
                 @"got lengths %d (key) and %d (value).",
                 cookie, [key length], [value length]];
           }
```

If we pass the two checks then we have found a good key-value pair and add it to the dictionary. We skip over the semicolon and any whitespace. This takes us to the first character of the next pair or to the end of the string.

```
           [self setObject:value forKey:key];

           [scn scanString:@";" intoString:NULL];
           [scn scanCharactersFromSet:space
               intoString:NULL];
        }
     }
  }
```

We return the dictionary when we are done scanning.

```
   return self;
}
```

The method `cookieToString` builds a set of `Set-Cookie` headers from the contents of the dictionary. It takes special care to process the cookie properties properly. There are two steps: first, processs all keys in the dictionary to build the cookie's properties as well as an array of key-value assignments and second, output a `Set-Cookie` header for each pair, using the properties from the first step.

The array `data` holds key-value pairs and the array `properties` holds properties. There is a boolean to indicate whether the cookie is marked secure or not. The variable `field` holds a single item being added to one of the arrays. We iterate over all keys of the dictionary.

```
- (NSString *)cookieToString
{
   NSMutableArray *data =
      [NSMutableArray arrayWithCapacity:1];
   NSMutableArray *properties =
      [NSMutableArray arrayWithCapacity:1];
   BOOL secure = NO;

   NSString *field;

   NSEnumerator *keyEnum = [self keyEnumerator];
   NSString *key;
   while((key=[keyEnum nextObject])!=nil){
```

The properties `domain` and `path` are easy: we build the key-value pair and add it to the array of properties.

```
      if([key isEqualToString:@"domain"]==YES ||
        [key isEqualToString:@"path"]==YES){
         field =
            [NSString stringWithFormat:@"%@=%@;",
                    key, [self objectForKey:key]];
         [properties addObject:field];
      }
```

The property `expires` is assumed to be a date object. We require a date string in the format that the protocol uses. We use a calendar format to obtain the result; note that we must use `GMT` for the time zone.

```
      else if([key isEqualToString:@"expires"]==YES){
        NSString *dateStr =
           [[self objectForKey:key]
              descriptionWithCalendarFormat:@"%a,␣%d-%b-%Y␣"
              @"%H:%M:%S␣GMT"
              timeZone:[NSTimeZone timeZoneWithName:@"GMT"]
              locale:nil];
         field =
            [NSString stringWithFormat:@"%@=%@;",
                    key, dateStr];
         [properties addObject:field];
      }
```

We set the boolean indicator for the property `secure` if it is among the keys in the dictionary. All other entries of the dictionary are assumed to be data, i.e. key-value pairs. We format each pair according to the requirements and store it in the array `data`. This completes the first step.

```
      else if([key isEqualToString:@"secure"]==YES){
         secure = YES;
      }
      else{
         field =
            [NSString stringWithFormat:@"%@=%@;",
                    key, [self objectForKey:key]];
         [data addObject:field];
      }
   }
```

We now have all the properties. This means that we can construct the property string, which stays the same for all pairs. We build a string that contains all properties separated by single spaces. If the property

secure has been set, than it is included at the end of the property string.

```
   NSString *props =
      [properties componentsJoinedByString:@" "];
   if(secure==YES){
      props = [props stringByAppendingString:@" secure"];
   }
```

The second step is to build the result string, which starts out empty. We iterate over all data lines and construct a Set-Cookie header by including first the data, and then the properties. We return the result when we are done.

```
   NSString *result = @"", *line;
   NSEnumerator *lineEnum = [data objectEnumerator];
   while((line=[lineEnum nextObject])!=nil){
      NSString *allFields =
         [NSString
            stringWithFormat:@"Set-Cookie: %@ %@\r\n",
            line, props];
      result =
         [result stringByAppendingString:allFields];
   }

   return result;
}
```

The remainder of this section shows how we test the code that we discussed above. Recall that our test program should add two new cookies each time it is invoked and display the cookies that it received and the time that remains until they expire.

We begin by obtaining the cookies that may have been passed in through HTTP_COOKIE. We also obtain a date that is thirty seconds in the future, or some other defined value. It determines the expiry of the new cookies that we are about to set.

```
#define NEWKEYS 2

int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   NSMutableDictionary *cookieValues =
      [[NSMutableDictionary alloc] initWithCookie];

   NSDate *date =
      [NSDate dateWithTimeIntervalSinceNow:PERSIST];
```

The new cookie set starts out containing two properties, the date and the path.

```
   NSMutableDictionary *newCookie =
      [NSMutableDictionary
         dictionaryWithObjectsAndKeys:date, @"expires",
         @"/cgi-bin/cookies.sh", @"path", nil];
```

We must now add the key-value pairs, NEWKEYS items to be precise. The key should be a string containing the character 'k', the last four digits of the time, and a three digit index to make it unique, and its value, most importantly, should be the current time, i.e. the time when it was created.

```
   long now = time(NULL); int k;
   srand48(now);
```

```
for(k=0; k<NEWKEYS; k++){
   NSString
      *key =
      [NSString stringWithFormat:@"k%04ld%03d",
              now%10000, k],
      *value =
      [NSString stringWithFormat:@"%ld",
              now];
   [newCookie setObject:value forKey:key];
}
```

We are now ready to output HTML code. There are two sections: a list of all cookies that we received and how much longer each will last, some auxiliary information and a button that invokes `cookies.sh`. The set of cookies comes first, followed by the header and the beginning of the body of the document.

```
printf("%s", [[newCookie cookieToString] cString]);
printf("Content-type: text/html\r\n\r\n");

printf("<HTML><HEAD><TITLE>Cookies</TITLE></HEAD><BODY>\n");
```

We prepare to iterate over the key-value pairs that we received. There is an enumerator and a counter. The data will not use HTML markup, so we output a PRE tag.

```
NSEnumerator *en;
NSString *key, *value;

int count = 1;
long rem;

puts("<PRE>");
```

First we print the data (key-value pairs) that we received; this helps to debug the program. We sort the keys in lexical order and prepare to iterate over the array thus obtained.

```
printf("Received:\n%s\n\n\n",
     [[cookieValues cookieToString] cString]);

en = [[[cookieValues allKeys]
        sortedArrayUsingSelector:@selector(compare:)]
       objectEnumerator];
while((key = [en nextObject])!=nil){
```

We lookup up the value associated to each key. The remaining time for this key is the difference between the time that has elapsed since its creation and the total persistence time. Note that differences between the clocks of the server and the client can cause the client to remove the cookie a bit sooner or a bit later than required. We construct a string that contains the key, the value, and the remaining time, print it and increment the counter.

```
   value = [cookieValues objectForKey:key];
   rem = (long)PERSIST-(now-atol([value cString]));

   NSString *item =
      [NSString stringWithFormat:@"key %d: %@\n"
              @"value %d: %@\n"
              @"remaining %d: %lds\n",
              count, key, count, value,
              count, rem];

   printf("%s\n", [item cString]);
```

```
        count++;
    }
```

We output an indicator phrase that shows whether a cookie was received. We also print the `Set-Cookie` headers that we sent earlier, so the user knows what cookies to expect when he submits the form.

```
    printf("%s receive a cookie.\n\n\n",
           ([cookieValues count]? "Did" : "Did not"));
    printf("%s\n", [[newCookie cookieToString] cString]);
```

The second section of the document contains a button that reloads the script, thereby forcing the client to send the current set of cookies to the server.

```
    puts("</PRE>");

    printf("<FORM ACTION=cookies.sh METHOD=GET>\n"
           "<INPUT TYPE=SUBMIT VALUE=Submit>\n"
           "</FORM>\n");

    printf("</BODY></HTML>\n");

    [pool release];
    exit(0);
}
```

## 5.4 Webchat with MySQL

*by Marko Riedel*

### 5.4.1 Idea

We use the recipes of the preceding sections and the MySQL C API available through `libmysqlclient` to build a web chat with MySQL. We suggest you acquaint yourself with `phpMyAdmin` and the `mysql` command line interface in order to facilitate your work with MySQL.

We will work with a database that stores two types of information: session data such as the user's name and how often her chat screen should be refreshed and the chat data, which consists of open and private channels that are protected by a password and the messages themselves.

The structure of the chat can be described by the three different screens it uses:

- the login screen: this is where the user enters her name, how often the chat display should be refreshed, and whether she wants to use an open or a private channel
- the channel selector screen: this screen lets the user start a new channel or join an existing one. This is also where the password for private channels is entered. There is a menu of current channels when the user selected open channels in the login screen; these show how many messages are available and when the channel was last active and are sorted with the most recently active channels first.
- the main chat screen: this is a frame set with the upper frame containing the latest messages from the channel and the bottom frame being a form where the user may enter her message.

There are a variety of error message screens.

The login sequence is as follows: the user identifies herself in the login screen, which causes a new cookie, i.e. session identifier with a certain expiry time to be sent, selects or starts a new channel and starts using the chat through the main screen. The program must handle all of these steps. Furthermore, it must bracket its actions by a read and write of the session data, which it reads into a dictionary whose keys are the individual items.

### 5.4.2 MySQL tables used

There are three tables: `session, channels` and `messages.` The first holds session data. There is one entry for each session initiated by a client. The second, i.e. the table `channels`, records channel properties and the the third, `messages`, records the actual messages. We now discuss the fields of each channel in detail. Note that the string '%d' in the create statements is a `printf`-like placeholder that will be replaced by a constant.

Fields of the table `session`:

```
CREATE TABLE session (
  sessid char(%d) NOT NULL,
  time BIGINT,
  login char(%d),
  chanid BIGINT,
  refresh INT,
  open ENUM('Y', 'N'),
  UNIQUE KEY (sessid),
  PRIMARY KEY (sessid)
)
```

- `sessid`: this is the session id, which is a string of alphanumeric characters that are not easy to guess
- `time`: the time when the session was last active, e.g. when the user sent a message, given in seconds since the epoch (Jan 1, 1970), as are all time stamps in this application
- `login`: the name that the user chose for herself
- `chanid`: the identifier of the channel that the user is currently logged in on
- `refresh`: this is the number of seconds after which the display frame of the main screen will refresh
- `open`: indicates whether the user is logged in to an open or a private channel

Fields of the table `channels`:

```
CREATE TABLE channels (
  id BIGINT NOT NULL AUTO_INCREMENT,
  time BIGINT NOT NULL,
  name char(%d),
  password char(%d) BINARY,
  open ENUM('Y', 'N'),
  PRIMARY KEY (id)
)
```

- `id`: is a unique identifier of the channel and assigned by MySQL
- `time`: the time of the most recent activity on the channel
- `name`: a string that contains the name of the channel, which we choose to restrict to alphanumeric characters
- `password`: the encrypted password for the channel if it is a private channel, the empty string otherwise
- `open`: a flag that indicates whether the channel is open or not

Fields of the table `messages`:

```
CREATE TABLE messages (
  time BIGINT NOT NULL,
  login char(%d),
  chanid BIGINT,
  msg BLOB,
  PRIMARY KEY (time)
)
```

- `time`: time this message was entered
- `login`: name of the user who entered the message
- `chanid`: identifier of the channel where the message was entered
- `msg`: the encoded text of the message, which the type `BLOB` restricts to $2^{16}$ characters, of which we only use a few, however, but more than fit into the `CHAR` type.

### 5.4.3 Preliminaries

There is a wrapper script that works like the wrappers in the recipes for working with cookies and form variables.

```
#! /bin/sh
#

export GNUSTEP_SYSTEM_ROOT=/usr/GNUstep
export TZ=Europe/Berlin
. $GNUSTEP_SYSTEM_ROOT/System/Makefiles/GNUstep.sh

/home/gnustep/webchat/shared_obj/ix86/linux-gnu/gnu-gnu-gnu/webchat
```

There is an important note concerning the makefile. We must make sure that the `libmysqlclient` library is linked after `gnustep-base`. The following makefile accomplishes this. (We have moved the code from the cookie and form recipes into the file `CGIaux.m`.)

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME=webchat
webchat_OBJC_FILES = webchat.m CGIaux.m

TARGET_SYSTEM_LIBS += -lmysqlclient
SHARED_CFLAGS     += -g

include $(GNUSTEP_MAKEFILES)/tool.make
```

### 5.4.4 Implementation I: auxiliary functions and definitions

Start by defining some constants to tune the behavior of the program. You are encouraged to experiment with these. The meaning of each constant follows.

- `CHATCGI`: this is the path to the CGI script and it tells the client that it should only send the session id cookie to this path
- `MYSQL_USER` and `MYSQL_PASSWORD`: these two define the MySQL user that carries out all database operations and it must of course be defined and have the right set of privileges; we assume furthermore that the database runs on the same host as the web server
- `CHATDB`: the name to use for the chat database
- `LEN_LOGIN, LEN_SESSID, LEN_CHANNEL, LEN_MSG` and `LEN_PASSWORD`: the maximum number of characters for the login name, the session id, the channel name, the message and the password, respectively
- `DISPLAY_MAX`: the maximum number of messages to display in the message display frame of the main screen
- `SESSPERSIST, CHANPERSIST, MSGPERSIST`: these are in seconds and determine the time that has to elapse for a session, channel or message to be deleted from the database.

```
#include <Foundation/Foundation.h>
#include <mysql/mysql.h>

#include "CGIaux.h"

#define CHATCGI "/cgi-bin/webchat.sh"

#define MYSQL_USER      "chatuser"
#define MYSQL_PASSWORD  "webchat"
#define CHATDB          "chatdb"

#define LEN_LOGIN           16
#define LEN_SESSID          16
#define LEN_CHANNEL         32
#define LEN_MSG            256
#define LEN_PASSWORD        32

#define DISPLAY_MAX         10

#define SESSPERSIST       10*60
#define CHANPERSIST (SESSPERSIST+5*60)
#define MSGPERSIST (CHANPERSIST+5*60)
```

A series of useful macros follows. The macro `MYSQL_ERR` turns the most recent error message from MySQL into an `NSString` object. The macros FMT1 to FMT6 help simplify the use of strings obtained from a format string, which occurs a lot in this application.

```
#define MYSQL_ERR  [NSString stringWithCString: \
                        mysql_error(&mysql)]

#define FMT1(_f, _a1) \
     [NSString stringWithFormat:_f, _a1]
#define FMT2(_f, _a1, _a2) \
     [NSString stringWithFormat:_f, _a1, _a2]
#define FMT3(_f, _a1, _a2, _a3) \
     [NSString stringWithFormat:_f, _a1, _a2, _a3]
#define FMT4(_f, _a1, _a2, _a3, _a4) \
     [NSString stringWithFormat:_f, _a1, _a2, _a3, _a4]
#define FMT5(_f, _a1, _a2, _a3, _a4, _a5) \
     [NSString stringWithFormat:_f, _a1, _a2, _a3, _a4, _a5]
#define FMT6(_f, _a1, _a2, _a3, _a4, _a5, _a6) \
     [NSString stringWithFormat:_f, _a1, _a2, _a3, _a4, _a5, _a6]
```

We now define a set of auxiliary functions that we will use in the `main` procedure of the script. The function `Empty` tests whether a string object is empty, i.e. whether it is `nil` or has length zero.

```
BOOL inline Empty(NSString *str)
{
   return (str==nil || ![str length]);
}
```

There are peculiarities pertaining to the way messages are being stored and retrieved that have to be taken into account. A message may contain characters that we would have to escape before the message can be given to MySQL. Furthermore, what if a message contains HTML tags? These could hamper the functionality of the message display screen. We avoid all these problems by encoding each byte of the message in two hexadecimal digits and store this safe representation. We reconstruct the original byte 'B'

when we decode the message and output it as the HTML entity '&#B;', which assures that even HTML tags will be displayed literally (as opposed to being interpreted as markup).

The function `EncodeMsg` iterates over each byte of the string and uses the upper and lower four bits as an index into a string of hexadecimal digits. It appends the two digits to the result. We are done when we have reached the end of the string. Of course this doubles the length of the string that we will store.

```
NSString *EncodeMsg(NSString *msg)
{
   NSString *result = @"";
   const char *ptr = [msg cString],
      *xdigits = "0123456789ABCDEF";

   while(*ptr){
      unsigned char item = *ptr++;
      result = [result stringByAppendingFormat:@"%c%c",
                  xdigits[item/16], xdigits[item%16]];
   }

   return result;
}
```

The function `DecodeMsgIntoEntities` also iterates over the individual bytes of its argument. It processes two bytes, i.e. digits, at a time and depends on digits and letters being in sequence in the ASCII string encoding. If it finds a decimal digit, then its value is given by the difference between the digit and the character '0'; the value of a hexadecimal digit is ten plus the difference between the digit and the character 'A'. It remains to compute the reconstructed byte and append the appropriate HTML entity to the result.

```
NSString *DecodeMsgIntoEntities(NSString *msg)
{
   NSString *result = @"";
   const char *ptr = [msg cString];

   while(ptr[0] && ptr[1]){
      int upper = (isdigit(ptr[0]) ?
                ptr[0]-'0' : ptr[0]-'A'+10);
      int lower = (isdigit(ptr[1]) ?
                ptr[1]-'0' : ptr[1]-'A'+10);
      result = [result stringByAppendingFormat:@"&#%d;",
                   upper*16+lower];
      ptr += 2;
   }

   return result;
}
```

The next two functions encapsulate two chunks of HTML that is output at several places in the program. The function `Footer` outputs a separator (line), followed by a link to the author's home page, followed by the name of the program, the host name and the current date. All of these should make it easy for the user to locate the server that is currently being used.

```
NSString *Footer()
{
   NSProcessInfo *pinfo = [NSProcessInfo processInfo];

   NSString *fstr =
      FMT3(@"<HR>\n"
          @"<ADDRESS>%@ by marko_riedel, "
          @"<A HREF="
```

```
            @"http://www.geocities.com/markoriedelde>\n"
            @"http://www.geocities.com/markoriedelde</A>\n"
            @"at␣%@,␣%@</ADDRESS>\n",
            [pinfo processName], [pinfo hostName],
            [[NSDate date] description]);

    return fstr;
}
```

There are several screens, including those for error messages, where the user should be given the chance to start over. This is what the output of the function `BackToLogin` does. It outputs a form containing a single button and an important hidden field: `cmd`, i.e. "command." This is set to "restart" and lets the user restart the session. We shall see later how commands are processed. In fact "restart" is not the only command; there is also "display" and "enter," which display the two frames of the main screen, respectively.

```
NSString *BackToLogin()
{
    NSString *btologin =
        @"<BR><FORM␣TARGET=␣top␣METHOD=POST␣"
        @"ACTION=webchat.sh>\n"
        @"<INPUT␣TYPE=HIDDEN␣NAME=cmd␣VALUE=restart>\n"
        @"<INPUT␣TYPE=SUBMIT␣"
        @"VALUE=\"Back␣to␣login\">\n"
        @"</FORM>\n";

    return btologin;
}
```

The function `ErrmsgAndExit` plays an important role in the program. We do extensive error checking (quite possibly a bit more than necessary) and we need a function that outputs an error message and lets the user start over. This is the purpose of `ErrmsgAndExit`. It is invoked with a short description of the error and a string that gives the details of the error that occurred. It also outputs the restart button.

```
void ErrmsgAndExit(NSString *desc, NSString *detail)
{
    NSString *body =
        FMT5(@"<HTML><HEAD>\n"
            @"<TITLE>%@</TITLE>\n"
            @"</HEAD><BODY␣BGCOLOR=white>\n"
            @"<H1>%@</H1>\n"
            @"%@\n%@\n%@\n"
            @"</BODY></HTML>\n",
            desc, desc, detail,
            BackToLogin(), Footer());

    printf("Content−type:␣text/html\r\n\r\n");
    printf("%s", [body cString]);

    exit(1);
}
```

One error condition that occurs quite frequently is when we expected an alphanumeric string and received something else, or that the length of the string is not what we require. The function `CheckForAlNum` tests for these conditions. First it verifies that the string has the right length and aborts with an error otherwise. Next it iterates over the characters that make up the string, testing each character with `isalnum` in turn. If it does not get to the terminating null byte then there was an illegal character. In this case it outputs a descriptive error message.

```
void CheckForAlNum(NSString *toCheck, NSString *desc,
                int min, int max)
{
   int len = (toCheck==nil ? 0 : [toCheck length]);
   NSString *err, *aux;

   if(len<min || len>max){
      err = FMT3(@"%@ empty or too long; "
               @"min %d, max %d characters.",
               desc, min, max);
      aux = FMT1(@"Got '%@'.", toCheck);
      ErrmsgAndExit(err, aux);
   }

   const char *ptr = [toCheck cString], *base = ptr;
   while(isalnum(*ptr++));
   if(*(ptr-1)){
      err = FMT1(@"Illegal character "
               @"in %@; must be alphanumeric.", desc);
      aux = FMT2(@"Got '&#%d' "
               @"at position %d.",
               (int)*(ptr-1), ptr-base);
      ErrmsgAndExit(err, aux);
   }
}
```

We now meet one of the most important functions of the program, i.e. `OutputPageAndExit`. Recall that we store the session data in the MySQL database. Whatever we do, the first step must be to read those session data into the session dictionary if present, and the last, to record the current contents of the dictionary. The read only occurs once, i.e. at the beginning of the program. The update may be invoked at different exit points of the program, which is why we put it into the function `OutputPageAndExit`.

There are two steps to this function: first, write the session data, and second, output the page and exit. The first step starts by building the MySQL query. It says to update the table `session` with the values from the session dictionary, which contains the session data that we retrieved from MySQL. Only the time is not set from a dictionary value. It is set to the current time, because that is the time of the most recent activity in the session.

```
void OutputPageAndExit(MYSQL mysql, NSMutableDictionary *session,
                  NSString *title, NSString *frames,
                  NSString *body, NSString *bodyargs)
{
   NSString *query =
      FMT6(@"UPDATE session SET "
          @"time = '%lu', login='%@', refresh='%@', "
          @"chanid = '%@', open='%@' "
          @"WHERE sessid='%@'",
          (long)[[NSDate date] timeIntervalSince1970],
          [session objectForKey:@"login"],
          [session objectForKey:@"refresh"],
          [session objectForKey:@"chanid"],
          [session objectForKey:@"open"],
          [session objectForKey:@"sessid"]);
```

The end of the first step is to actually do the query and generate an error message if it fails.

```
   if(mysql_query(&mysql, [query cString])){
      NSString *err =
```

```
            FMT1(@"Couldn't write session data: %@", query);
        ErrmsgAndExit(err, MYSQL_ERR);
    }
```

Step two is easy: output the page as described by the arguments. We include the footer. The argument `bodyargs` can be used to set BODY properties, such as an "onLoad" that starts a timer or sets the focus.

```
    printf("Content-type: text/html\r\n");
    printf("Pragma: no-cache\r\n\r\n");

    printf("<HTML><HEAD><TITLE>%s</TITLE>\n%s</HEAD>\n",
          [title cString], [frames cString]);
    printf("<BODY%s>\n", [bodyargs cString]);
    printf("%s", [body cString]);
    printf("%s\n", [Footer() cString]);
    printf("</BODY></HTML>\n");

    exit(0);
}
```

The function `SetCookie` is as important as `OutputPageAndExit`. There are three cases where it must be used. We must generate and output a new cookie if the script did not receive one, which signals the start of a new session. We should reset the expiry time of the cookie when the session is restarted. Finally, the expiry time must also be updated when the user enters a message, which indicates activity on the session.

The expiry time of the cookie should be `SESSPERSIST` seconds from now. We build the cookie dictionary with the expiry time, the path and most importantly, the session id, convert it to a string, then to a C string, and output the result.

```
void SetCookie(NSString *sessid)
{
    NSDate *expDate =
        [NSDate dateWithTimeIntervalSinceNow:SESSPERSIST];

    NSMutableDictionary *cdict =
        [NSMutableDictionary
            dictionaryWithObjectsAndKeys:sessid, @"SESSID",
            expDate, @"expires", @CHATCGI, @"path", nil];

    printf([[cdict cookieToString] cString]);
}
```

It remains to discuss two auxiliary functions that encapsulate MySQL queries. The first of these, `FirstFieldsFromResult`, takes a MySQL result consisting of some number of rows, of which we are only interested in the first field. It produces an array of those fields. It fetches the rows in turn, converts the C string in the first field to an `NSString` and stores the latter in an array, which it returns when it has processed all the rows.

```
NSMutableArray *FirstFieldsFromResult(MYSQL_RES *result)
{
    MYSQL_ROW row;
    NSMutableArray *data =
        [NSMutableArray arrayWithCapacity:1];

    while((row=mysql_fetch_row(result))!=NULL){
        NSString *fstr =
            [NSString stringWithCString:row[0]];
        [data addObject:fstr];
    }
```

```
    return data;
}
```

The second auxiliary function for use with MySQL is `RowsFromQuery`, which also processes rows resulting from a query. It returns an array of dictionaries, one dictionary per row. The keys of the dictionary are the names of the fields, and the values are the actual data. First we make the query and signal any errors that may have occurred.

```
NSMutableArray *RowsFromQuery(MYSQL mysql, NSString *query)
{
    if(mysql_query(&mysql, [query cString])){
        NSString *err = FMT1(@"Query failed: %@", query);
        ErrmsgAndExit(err, MYSQL_ERR);
    }
```

Next we obtain the query result so that we may fetch the rows, again checking for possible errors.

```
    MYSQL_RES *result;
    if((result=mysql_use_result(&mysql))==NULL){
        NSString *err = FMT1(@"No results for query: %@", query);
        ErrmsgAndExit(err, MYSQL_ERR);
    }
```

We need the field names, which are the keys to the row dictionaries and need only be fetched once. We obtain the number of fields and a pointer to the fields themselves. We convert each field name to an `NSString` and add it to the array of field names. The name of an element at position $k$ of a row is given by the name at position $k$ in the array of field names.

```
    unsigned int ftotal = mysql_field_count(&mysql), fcur;
    MYSQL_FIELD *fields = mysql_fetch_fields(result);
    NSString *fstr;

    NSMutableArray
        *fnames = [NSMutableArray arrayWithCapacity:ftotal];
    for(fcur=0; fcur<ftotal; fcur++){
        fstr = [NSString stringWithCString:fields[fcur].name];
        [fnames addObject:fstr];
    }
```

We can process the rows now that we have the field names. We fetch the total number of rows and initialize a mutable array having this capacity. It will hold dictionaries.

```
    MYSQL_ROW row;
    my_ulonglong rtotal = mysql_num_rows(result);

    NSMutableArray
        *data = [NSMutableArray arrayWithCapacity:rtotal];
```

Now iterate over the rows. Create a dictionary with the right capacity for each row. Then iterate over the fields, storing them as `NSString` objects. The key is obtained from the corresponding entry in the array of field names. Add the row dictionary to the data array once all the fields have been processed.

```
    while((row=mysql_fetch_row(result))!=NULL){
        NSMutableDictionary *rowData =
            [NSMutableDictionary dictionaryWithCapacity:ftotal];
        for(fcur=0; fcur<ftotal; fcur++){
            fstr = [NSString stringWithCString:row[fcur]];
            [rowData setObject:fstr
```

```
                forKey:[fnames objectAtIndex:fcur]];
      }
      [data addObject:rowData];
   }
```

It remains to free the MySQL result and return the array of dictionaries.

```
   mysql_free_result(result);

   return data;

}
```

This is the end of the first section.

### 5.4.5 Implementation II: `main`

We are now ready to discuss the function `main`, which does the actual work of generating the different screens and interacting with the database. We declare commonly used variables that hold a MySQL query, an error message and an error message description, repectively. The first thing to do is to initialize the MySQL client library and connect to the local host as user `MYSQL_USER`, with the password `MYSQL_PASSWORD`. We send any errors to the client, which is how all errors will be handled.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   NSString *query, *err, *aux;
   MYSQL mysql;
   MYSQL_RES *result;

   mysql_init(&mysql);

   if(mysql_real_connect(&mysql,"localhost",
                  MYSQL_USER, MYSQL_PASSWORD,
                  NULL, 0, NULL, 0)==NULL){
      ErrmsgAndExit(@"MySQL connect failed", MYSQL_ERR);
   }
```

We ask for the list of databases that match the regular expression `CHATDB` and process any errors. We convert the result into an array of database names and free the MySQL result.

```
   if((result=mysql_list_dbs(&mysql, CHATDB))==NULL){
      ErrmsgAndExit(@"Couldn't list databases", MYSQL_ERR);
   }
   NSMutableArray *databases = FirstFieldsFromResult(result);
   mysql_free_result(result);
```

We have a problem if there was no database that matched the pattern. We try to create it if this is the case and abort with an error message if the create query failed.

```
   if([databases containsObject:@CHATDB]==NO){
      query = [NSString stringWithFormat:@"CREATE DATABASE %s",
                  CHATDB];
      if(mysql_query(&mysql, [query cString])){
         err = FMT1(@"Couldn't create database: %s", CHATDB);
         ErrmsgAndExit(err, MYSQL_ERR);
      }
   }
```

We can select the chat database once we know that it exists.

```
if(mysql_select_db(&mysql, CHATDB)){
    err = FMT1(@"Couldn't select database: %s", CHATDB);
    ErrmsgAndExit(err, MYSQL_ERR);
}
```

We need to verify that the three tables `session`, `channel` and `messages` are present. Hence we ask MySQL for a list of tables. We may use `FirstFieldsFromResult` because every row of the result contains exactly one field.

```
if((result=mysql_list_tables(&mysql, NULL))==NULL){
    ErrmsgAndExit(@"Couldn't list tables", MYSQL_ERR);
}
NSMutableArray *tables = FirstFieldsFromResult(result);
mysql_free_result(result);
```

What follows is a very important declaration. We declare an array that holds the names of the tables that must be present, and a second array containing `CREATE` statements for each table for use if the table is not present. The third array holds the maximum age of an entry for each table. These three arrays use the same order.

```
NSString *tnames[] = { @"session", @"channels", @"messages", nil };
NSString *tcreate[] = {
    FMT2(@"CREATE TABLE session ("
        @"sessid char(%d) NOT NULL,"
        @"time BIGINT,"
        @"login char(%d),"
        @"chanid BIGINT,"
        @"refresh INT,"
        @"open ENUM('Y','N'),"
        @"UNIQUE KEY (sessid),"
        @"PRIMARY KEY (sessid)"
        @")", LEN_SESSID, LEN_LOGIN),
    FMT2(@"CREATE TABLE channels ("
        @"id BIGINT NOT NULL AUTO_INCREMENT,"
        @"time BIGINT NOT NULL,"
        @"name char(%d),"
        @"password char(%d) BINARY,"
        @"open ENUM('Y','N'),"
        @"PRIMARY KEY (id)"
        @")", LEN_CHANNEL, LEN_PASSWORD),
    FMT1(@"CREATE TABLE messages ("
        @"time BIGINT NOT NULL,"
        @"login char(%d),"
        @"chanid BIGINT,"
        @"msg BLOB,"
        @"PRIMARY KEY (time)"
        @")", LEN_LOGIN)
};
long persists[] = {
    SESSPERSIST,
    CHANPERSIST,
    MSGPERSIST
};
```

We iterate over these three arrays in parallel. If a table is not present in the list of tables that we obtained from MySQL, then it must be created, which we immediately try to do, checking for errors as we go.

```
int tindex = 0;

while(tnames[tindex]!=nil){
   if([tables containsObject:tnames[tindex]]==NO){
      if(mysql_query(&mysql, [tcreate[tindex] cString])){
         err = FMT1(@"Query failed: %@", tcreate[tindex]);
         ErrmsgAndExit(err, MYSQL_ERR);
      }
   }
```

The second half of the loop does what we may call house-keeping. It erases records that are too old. Errors are sent to the client. (We probably should have put the `DELETE` in an *else* clause, because there is nothing to delete from a new table, although it can do no harm.)

```
   long now =
      [[NSDate date] timeIntervalSince1970];
   query = FMT2(@"DELETE FROM %@ WHERE time<%ld",
            tnames[tindex], now-persists[tindex]);
   if(mysql_query(&mysql, [query cString])){
      err = FMT1(@"Delete failed: %@", query);
      ErrmsgAndExit(err, MYSQL_ERR);
   }
   tindex++;
}
```

We are now ready to do some actual processing. We parse any GET or POST variables that were passed to the script. We also parse the cookie, if there was one, and look up the session id in the cookie dictionary.

```
NSMutableDictionary *params =
   [[NSMutableDictionary alloc] initWithCGI];
NSString *cmd = [params objectForKey:@"cmd"];

NSMutableDictionary *cvals =
   [[NSMutableDictionary alloc] initWithCookie];
NSString *sessid = [cvals objectForKey:@"SESSID"];
```

If there was no session id in the cookie then we are either starting a new session or we are being called to display a portion of the main screen, but the cookie has expired. We send an explanatory error message in the latter case.

```
if(sessid==nil){
   if(cmd!=nil &&
     [cmd isEqualToString:@"display"]==YES ||
     [cmd isEqualToString:@"enter"]==YES){
      err = FMT1(@"Session timed out, date is %@.",
               [[NSDate date] description]);
      aux = FMT1(@"Command was: %@", cmd);
      ErrmsgAndExit(err, aux);
   }
```

If there was no timeout but there is no session id, then we must generate and initialize a new one that should preferably not be easily guessable, so that no user can join someone else's session by guessing an actual value of a session id that is in use and sending it as a cookie. We seed the random number generator and generate each character of the key in turn, asking for a random number from the interval $[0, 26+26+10-1]$. Values from $[0, 26-1]$ yield lower case letters, values from $[26, 26+26-1]$, upper case letters and values from $[26+26, 26+26+10-1]$, digits. We append the character obtained in this manner to the session id.

```
   int pos;
```

```
    srand48(time(NULL));
    sessid = @"";

    for(pos=0; pos<LEN_SESSID; pos++){
       char item;
       int choice = lrand48() % (26+26+10);
       item = (choice < 26 ? 'a'+choice :
             (choice< 52 ? 'A'+choice−26 :
              '0'+choice−26−26));
       sessid =
          [sessid stringByAppendingFormat:@"%c", item];
    }
```

But it is not enough merely to generate the key, we must also store it in the sessions table and initialize the remaining fields of the session. We use an `INSERT` query for this purpose. It stores the session id and the current time, leaving the other fields empty. Errors are sent to the client; e.g. the query will fail in the admittedly unlikely case that we generated a session id that is already in the table, because we declared the corresponding field to be unique. We are done initializing once we output the cookie for storage in the client with the session id using `Set-Cookie`.

```
    query = FMT2(@"INSERT INTO session (sessid, time) "
              @"VALUES ('%@', '%ld')",
              sessid,
              (long)[[NSDate date] timeIntervalSince1970]);
    if(mysql_query(&mysql, [query cString])){
       err = FMT1(@"Couldn't initialize session: %@"
                @"<BR>Try again.", query);
       ErrmsgAndExit(err, MYSQL_ERR);
    }

    SetCookie(sessid);
}
```

We may now be certain that we have a valid session id, even if it has only just been generated. Hence we may read the session data into a dictionary. This is done with a `SELECT` query that selects all fields from those records that match the session id. There should be exactly one such record; everything else is an error. We extract the session dictionary.

```
NSString *title, *frames, *body, *bodyargs;

query = FMT1(@"SELECT * FROM session WHERE sessid='%@'",
          sessid);
NSMutableArray *sessdata = RowsFromQuery(mysql, query);
if([sessdata count]!=1){
   err = FMT2(@"Found %d entries for %@.",
            [sessdata count], sessid);
   ErrmsgAndExit(err, query);
}
NSMutableDictionary *session = [sessdata objectAtIndex:0];
```

We must respond to the command "restart" before we do any additional processing. We clear all fields in the session dictionary and output the cookie, which will now last an additional `SESSPERSIST` seconds. Remember that the session time stamp will be set by `OutputPageAndExit`.

```
if([cmd isEqualToString:@"restart"]){
   [session setObject:@"" forKey:@"login"];
   [session setObject:@"" forKey:@"refresh"];
```

```
    [session setObject:@"" forKey:@"open"];
    [session setObject:@"" forKey:@"chanid"];
    SetCookie(sessid);
}
```

We have now come to the point where we output the first of our several screens, which is the login screen. We output the login form if there is no user name in the session dictionary and we did not receive one via a GET or POST. The form contains a table with three rows, i.e. name, refresh and session type. There is a submit button.

```
if(Empty([session objectForKey:@"login"]) &&
  Empty([params objectForKey:@"login"])){
   title = @"Chat login";
   frames = @"";
   body = @"<FONT SIZE=24><B>Login</B></FONT>\n"
      @"<FORM METHOD=POST ACTION=webchat.sh>\n"
      @"<TABLE>\n"
      @"<TR><TD>Name:</TD>\n"
      @"<TD><INPUT TYPE=TEXT NAME=login>\n"
      @"</TD></TR>\n"
      @"<TR><TD>Refresh (in s)</TD>\n"
      @"<TD>\n"
      @"<INPUT TYPE=TEXT NAME=refresh>\n"
      @"</TD></TR>\n"
      @"<TR><TD>Session type:</TD>\n"
      @"<TD>\n"
      @"<SELECT NAME=open>\n"
      @"<OPTION SELECTED VALUE=Y>  open </OPTION>\n"
      @"<OPTION VALUE=N> private </OPTION>\n"
      @"</SELECT>\n"
      @"</TD></TR>\n"
      @"</TABLE>\n"
      @"<INPUT TYPE=SUBMIT VALUE=Continue>\n"
      @"</FORM>\n";
   bodyargs = @"";
   OutputPageAndExit(mysql, session,
               title, frames, body, bodyargs);
}
```

The remaining screens require a time stamp, so we set it here for use in the rest of the program.

```
long now =
   [[NSDate date] timeIntervalSince1970];
```

The first case was an empty session and no CGI variables, which lead to the login screen. The second case is an empty session and the appropriate CGI variables. We set the flag `readParams` in this case and copy the variables from the CGI dictionary to the session dictionary.

```
BOOL readParams = NO;
if(Empty([session objectForKey:@"login"])){
   readParams = YES;
   [session setObject:[params objectForKey:@"login"]
          forKey:@"login"];
   [session setObject:[params objectForKey:@"refresh"]
          forKey:@"refresh"];
   [session setObject:[params objectForKey:@"open"]
          forKey:@"open"];
}
```

We may now read the settings that correspond to the data obtained from the login screen.

```
NSString *login = [session objectForKey:@"login"];
int refresh = [[session objectForKey:@"refresh"] intValue];
char open = ([[session objectForKey:@"open"]
             isEqualToString:@"Y"] ? 'Y' : 'N');
```

There is error checking to do if we obtained those settings from the CGI dictionary. We verify that the login name is not empty and consists of alphanumeric characters only. The refresh should be set to a positive interval.

```
if(readParams==YES){
   CheckForAlNum(login, @"Login", 1, LEN_LOGIN);

   if(refresh<1){
      err = @"Refresh must be positive.";
      aux = FMT1(@"Got '%d'.", refresh);
      ErrmsgAndExit(err, aux);
   }
}
```

We now have the data that correspond to the login screen. The channel screen is next. We generate this screen if the channel id has not been set and is not among the CGI parameters.

```
NSString *chanid = [session objectForKey:@"chanid"];
int numericCID = [chanid intValue];

if(!numericCID && Empty([params objectForKey:@"channel"])){
```

We will generate the output line by line or item per item in all cases that follow and only join those lines when everything has been generated. We do not send HTML as it is created. This is so that there is no mixing between regular output and error messages.

There are two cases that correspond to open and private channels. Both contain forms. They have in common that the first form contains a text input where the name of the channel may be entered. This can be used to select an existing channel or to create a new one.

```
   NSString *line;
   NSMutableArray *lines =
      [NSMutableArray arrayWithCapacity:8];

   title = @"Chat login: select or start channel";
   body = @"<FONT SIZE=24>\n"
      @"<B>Login: select or start channel</B>\n"
      @"</FONT>\n"
      @"<FORM METHOD=POST ACTION=webchat.sh>\n"
      @"Channel: <INPUT TYPE=TEXT NAME=channel>\n";
```

Now is where the two cases (open vs. private) diverge. We output a password field when the user has selected private channels. The submit button is again common to both cases.

```
   if(open=='N'){
      line = @"Password: <INPUT TYPE=PASSWORD NAME=password>";
      [lines addObject:line];
   }
   line = @"<INPUT TYPE=SUBMIT VALUE=Enter>\n</FORM>";
   [lines addObject:line];
```

We wish to present a menu of currently open channels when the user has selected open channels. We find those channels with a `SELECT` query. We group the messages by channel id and select the channel name,

the number of messages and the newest time stamp from the table `channels`, which is updated every time a message is recorded. The where clause binds the channel name from the table `channels` to the channel id from the table `messages`. The result is a set of rows with three fields, that hold the channel name, the number of messages for the channel, and the newest message time stamp for the channel. We must check the `open` property so that our menu will not display private channels. We let MySQL do the sorting. The result will list channels according to their time stamps, with the channel that most recently had any activity first.

```
if(open=='Y'){
   query = @"SELECT␣"
      @"name,␣"
      @"COUNT(chanid),␣"
      @"channels.time␣"
      @"FROM␣channels,␣messages␣"
      @"WHERE␣channels.id=messages.chanid␣"
      @"AND␣open='Y'␣"
      @"GROUP␣BY␣messages.chanid␣"
      @"ORDER␣BY␣channels.time␣DESC";
   NSMutableArray *chanData =
      RowsFromQuery(mysql, query);
   NSEnumerator *chanEnum =
      [chanData objectEnumerator];
   NSMutableDictionary *channel;
```

We output the channel menu as a table, iterating over the channels with the enumerator.

```
   line = @"<TABLE␣BORDER␣BGCOLOR=white>\n";
   [lines addObject:line];

   while((channel = [chanEnum nextObject])!=nil){
```

Every entry in the menu is a form that contains a single button, which selects the respective channel. We must convert the time stamp into a readable date format, which we do with `NSDate`'s `description`. The number of messages is given by the appropriate field from the query.

```
      [lines addObject:
              @"<FORM␣METHOD=POST␣ACTION=webchat.sh"];

      long tval =
         atol([[channel objectForKey:@"time"]
                cString]);
      NSString *dateStr =
         [[NSDate dateWithTimeIntervalSince1970:tval]
            description];
      NSString *count =
         [channel
            objectForKey:@"COUNT(chanid)"];
```

An entry in the menu has three fields: the submit button, the time stamp, and the number of messages. The submit button is labeled with the name of the channel. This completes the form for the current entry, which we also could have put inside a single "TD" item rather than wrapping the entire row in the form.

```
      line = FMT4(@"<TR><TD>\n"
               @"<INPUT␣TYPE=SUBMIT␣"
               @"NAME=channel␣"
               @"VALUE=%@></TD>\n<TD>"
               @"%@</TD>\n<TD>␣%@␣message%@␣"
               @"</TD></TR>",
               [channel objectForKey:@"name"],
```

```
                    dateStr, count,
                    ([count isEqualToString:@"1"]==NO ?
                     @"s" : @""));
            [lines addObject:line];

            [lines addObject:@"</FORM>"];
        }
```

The special case of the open chanel menu ends with a closing tag for the table. The last step is to output everything and exit the program. This step is again common to open and private channels.

```
        line = @"</TABLE>\n";
        [lines addObject:line];
    }

    frames = @"";
    body =
        [body stringByAppendingString:
                [lines componentsJoinedByString:@"\n"]];
    bodyargs = @"";
    OutputPageAndExit(mysql, session,
                title, frames, body, bodyargs);
}
```

If we have got this far, but do not have a channel id, then the user must have submitted a channel name and possibly a password using precisely the form whose generation we just described. Hence we extract the channel name and the password form the dictionary of CGI parameters. There are two cases: either the channel exists already, and we select it, otherwise, we create a new channel and select it. (We pre-declare some variables that will hold the dictionary for the current channel and its name.)

```
NSMutableArray *chanLookup;
int chanCount;
NSMutableDictionary *chanRecord;
NSString *channel;

if(!numericCID){
    NSString *candidate = [params objectForKey:@"channel"];
    NSString *password = [params objectForKey:@"password"];
```

We will only accept channel names and passwords that are alphanumeric and have the right length.

```
    CheckForAlNum(candidate, @"Channel", 4, LEN_CHANNEL);
    if(open=='N'){
        CheckForAlNum(password, @"Password", 4, LEN_PASSWORD);
    }
```

We need to know whether this is a new channel or not, so we send the appropriate query to MySQL, using the channel name to select matching channels. We have a duplicate channel if we get more than one record with this name. (This should not happen.)

```
    query = [NSString stringWithFormat:@"SELECT "
                    @"id, open, name "
                    @"FROM channels WHERE name='%@' ",
                    candidate];

    chanLookup = RowsFromQuery(mysql, query);
    chanCount = [chanLookup count];

    if(chanCount>1){
```

```
        err = FMT1(@"Duplicate channel: %@.", candidate);
        aux = FMT1(@"Count was %d.", chanCount);
        ErrmsgAndExit(err, aux);
    }
```

If we obtained a single record, then the user is trying to join an existing channel, in which case we must verify that the user's choice of an open or private channel matches the setting in the record for the selected channel and signal an error otherwise.

```
    else if(chanCount==1){
        chanRecord = [chanLookup objectAtIndex:0];

        if([[chanRecord objectForKey:@"open"]
              isEqualToString:(open=='Y' ? @"Y" : @"N")]==NO){
            err = FMT1(@"Channel is not %@.",
                    (open=='Y' ? @"open" : @"private"));
            ErrmsgAndExit(err, candidate);
        }

        channel = [chanRecord objectForKey:@"name"];
        chanid = [chanRecord objectForKey:@"id"];
    }
```

We have a new channel if we did not obtain any records from the `SELECT` query. We build an `INSERT` query that contains the time stamp, the name, the open flag and the encrypted password. (The salt will be stored in the first two characters of the output from `ENCRYPT`.)

```
    else{
        query = FMT4(@"INSERT INTO "
                  @"channels "
                  @"(id, time, name, password, open) "
                  @"VALUES ('', '%ld', '%@', "
                  @"ENCRYPT('%s'), '%c')",
                  now, candidate,
                  (Empty(password)==YES ?
                   "" : [password cString]), open);
```

The channel was created if the query succeeded. We can set the channel name to the candidate value. We must ask MySQL for the channel id that it created. (Recall that `AUTO_INCREMENT` was set.)

```
        if(mysql_query(&mysql, [query cString])){
            err = FMT1(@"Couldn't create channel: %@",
                    candidate);
            ErrmsgAndExit(err, MYSQL_ERR);
        }

        channel = candidate;
        chanid = FMT1(@"%ld", (long)mysql_insert_id(&mysql));
    }
```

At this point we can be certain of having a valid channel id and a channel name. It remains to check the password in the case of a logon to a private channel that wasn't just created (`chanCount==1`). We ask MySQL for the id of any channel whose encrypted password matches the result of encrypting the user-supplied password, with the first two characters of the stored password being the salt.

```
    if(open=='N' && chanCount==1){
        query = FMT2(@"SELECT "
                  @"id FROM channels "
                  @"WHERE id = '%@' AND "
```

```
                @"password = "
                @"ENCRYPT('%@', LEFT(password, 2))",
                chanid, password);
```

The query must have succeeded and there must be a result set for us to process if we are to proceed. We could have used `RowsFromQuery`, except that we are not interested in the particular record, but rather in the number of records. There should be a single record.

```
        if(mysql_query(&mysql, [query cString])){
            err = FMT1(@"Password query failed: %@", query);
            ErrmsgAndExit(err, MYSQL_ERR);
        }
        if((result=mysql_store_result(&mysql))==NULL){
            err = FMT1(@"No result from password query: %@",
                    query);
            ErrmsgAndExit(err, MYSQL_ERR);
        }
```

We obtain the number of rows. There will be no rows if the password was wrong and one row otherwise. We send an error ("Permission denied.") to the client if that was the case. This concludes the processing associated to channel selection.

```
        int rescount = (int)mysql_num_rows(result);
        if(rescount!=1){
            aux = FMT6(@"User was %@, "
                    @"channel %@, id %@,"
                    @"password %@; "
                    @"combo gave %d entries.<P>"
                    @"query was: %@",
                    login, channel, chanid,
                    password, rescount, query);
            ErrmsgAndExit(@"Permission denied.", aux);
        }
        mysql_free_result(result);
    }
```

The following `else` clause deals with the case where we had a valid channel id in the session dictionary. We need the channel's name rather than its id for the display frame, which should not only display messages, but also indicate what user is viewing what channel. Hence we build a query that will yield the channel's name and execute it.

```
    else{
        query = FMT1(@"SELECT name FROM channels WHERE id='%@'",
                chanid);
        chanLookup = RowsFromQuery(mysql, query);
        chanCount = [chanLookup count];
```

There is something wrong if we obtained no entry or more than one entry. We set the channel name if everything was okay.

```
        if(chanCount!=1){
            err = FMT1(@"Expected one channel for ID %@.",
                    chanid);
            aux = FMT1(@"Got %d.", chanCount);
            ErrmsgAndExit(err, aux);
        }

        channel = [[chanLookup objectAtIndex:0]
                   objectForKey:@"name"];
```

```
}
```

If we got this far in the program, then the session must already be fully initialized: we have a user name, a refresh setting, an "open" flag and a channel id and a channel name. We should output either the main chat screen (frame set), the upper frame (message display) or the lower frame (message entry form). What exactly we should do is determined by the parameter "cmd." It is either "display" or "enter" or nil. Anything else is an error. We start by responding to the display command. We select the message text and the author's name from the database. We again let MySQL do the sorting.

```
if(cmd!=nil){
   if([cmd isEqualToString:@"display"]==YES){
      query = FMT2(@"SELECT msg,login "
                @"FROM messages "
                @"WHERE chanid='%@' "
                @"ORDER BY time "
                @"DESC LIMIT %d",
                chanid, DISPLAY_MAX);
      NSMutableArray *msgs = RowsFromQuery(mysql, query);
```

We emit the refresh header right away. This is safe to do because we know that there cannot be any errors after this point. We initialize the parameters for `OutputPageAndExit`. The first line of the body displays the current user and the channel name. We set the "onLoad" event property to a Javascript timeout, which is in milliseconds. This is to be on the safe side; theoretically, either the refresh header or the timeout alone should suffice.

```
      printf("Refresh: %d\r\n", refresh);
      title = @"Chat message";
      frames = @"";

      body = FMT2(@"<FONT SIZE=24>\n"
               @"<B>User: %@   Channel: %@</B>\n"
               @"</FONT>\n<P>\n", login, channel);
      bodyargs = FMT1(@" onLoad=\"javascript:"
                  @"window.setTimeout("
                  @"'location.reload() ', %ld)\"",
                  (long)(1000)*(long)refresh);
```

The messages go into a HTML table. We build the table line by line, storing lines in the array `msgOutput`. We obtain an iterator for the set of messages and loop over it.

```
      NSMutableArray *msgOutput =
         [NSMutableArray
            arrayWithObject:
               @"<TABLE WIDTH=100% BORDER>"];
      NSEnumerator *msgEnum = [msgs objectEnumerator];
      NSMutableDictionary *item;
      while((item = [msgEnum nextObject])!=nil){
```

The content of the message as retrieved from MySQL consists of a string of hexdecimal values, which we decode into HTML entities. There is one row with two columns for each message. The first column holds the name of the author and the second the message itself. We store the string for the current row in the output array.

```
         NSString *encoded = [item objectForKey:@"msg"],
            *entities = DecodeMsgIntoEntities(encoded);
         NSString *row =
            FMT2(@"<TR><TD WIDTH=20%>"
                @"%@</TD><TD WIDTH=80%>%@</TR>",
                [item objectForKey:@"login"],
```

```
                entities);
            [msgOutput addObject:row];
        }
```

We close the table once we have iterated over all messages and produce a single string by joining the output lines, separating them from each other with a newline character. We are now done processing the "display" command and may output the page.

```
        [msgOutput addObject:@"</TABLE>"];

        NSString *allmsgs =
            [msgOutput componentsJoinedByString:@"\n"];
        body = [body stringByAppendingString:allmsgs];

        OutputPageAndExit(mysql, session,
                    title, frames,
                    body, bodyargs);
    }
```

The second command that we must implement is the command "enter," which records a message and update's the channel's time stamp. There are two steps. If there is a message among the CGI parameters, then we must record it. We must also output a message submit form thereafter, regardless of whether there was a message or not.

In case of a message we encode it as a string of hexadecimal values first. Then we build the query, which records the time, the author, the channel id and the text of the message.

```
    else if([cmd isEqualToString:@"enter"]==YES){
        NSString *msg = [params objectForKey:@"msg"];
        if(!Empty(msg)){
            NSString *encoded = EncodeMsg(msg);
            query = FMT4(@"INSERT "
                        @"INTO messages "
                        @"(time, login, chanid, msg) "
                        @"VALUES"
                        @"('%ld', '%@', '%@', '%@')",
                        now, login, chanid, encoded);
```

The query must succeed or we output an error message.

```
            if(mysql_query(&mysql, [query cString])){
                err = FMT1(@"Couldn't insert message: %@",
                          DecodeMsgIntoEntities(encoded));
                ErrmsgAndExit(err, MYSQL_ERR);
            }
```

The process of recording a message includes an update of the channel's time stamp to indicate that there has been activity on the channel. Recall that this time stamps determines how long the channel will stay in the database and where on the channel menu it will be displayed if it is an open channel. The query is simple: enter the time stamp in the record with the appropriate channel id.

```
            query = FMT2(@"UPDATE channels "
                        @"SET time='%ld'"
                        @"WHERE id='%@'", now, chanid);
```

This query, too, must succeed, or we signal an error.

```
            if(mysql_query(&mysql, [query cString])){
                NSString *err =
                    FMT2(@"Couldn't update "
```

```
                    @"channel: %@, query %@",
                    channel, query);
                ErrmsgAndExit(err, MYSQL_ERR);
            }
        }
```

Step two of the response to the command "enter" is to output the message entry form. This is mostly static text and includes the button that takes the user back to the login screen. This button should not be missing from the main screen. It remains to point out the Javascript focus call, which makes it possible for a user to send a sequence of messages without having to click in the message field every time. There are three fields: the message, a hidden field which sets the command to "enter" and a submit button.

```
        title = @"Chat message";
        frames = @"";
        body = FMT1(@"<FONT SIZE=24>"
                 @"<B>Message</B></FONT>\n"
                 @"<FORM NAME=enter METHOD=POST "
                 @"ACTION=webchat.sh>\n"
                 @"<INPUT TYPE=TEXT NAME=msg SIZE=80 "
                 @"MAXLENGTH=%d>\n"
                 @"<INPUT TYPE=HIDDEN NAME=cmd "
                 @"VALUE=enter>\n"
                 @"<INPUT TYPE=SUBMIT VALUE=Submit>\n"
                 @"</FORM>\n", LEN_MSG);
        body = [body stringByAppendingString:BackToLogin()];
        bodyargs = @" onLoad="
           @"\"javascript:document.enter.msg.focus();\"";
```

We must reset the cookie's expiry date because an enter signals that there has been activity for the current session. We invoke `SetCookie` for this purpose, and conclude by outputting the page.

```
        SetCookie(sessid);
        OutputPageAndExit(mysql, session,
                     title, frames, body, bodyargs);
    }
```

We have now responded to all possible commands. Any other command is an error.

```
    else{
        err = @"Unknown command.";
        aux = [NSString stringWithFormat:@"Got %@.",
                    cmd];
        ErrmsgAndExit(err, aux);
    }
}
```

The final section of the program is for the case when all session variables were set, but no command was received. This is the default case and it indicates that we should display the main screen with its two frames. The frame set allocates 65% of the browser's height to the upper frame, where messages are displayed, and 35% to the lower frame, where messages are entered. The request method by which commands are transmitted is GET. We can output the page once the arguments for `OutputPageAndExit` have been initialized. The exit statement at the end of the program is not reached.

```
title = FMT2(@"Chat user: %@ Channel: %@",
          login, channel);
frames = @"<FRAMESET ROWS=\"65%, 35%\">\n"
    @"<FRAME SRC=webchat.sh?cmd=display "
    @"SCROLLING=no NAME=display>\n"
    @"<FRAME SRC=webchat.sh?cmd=enter "
```

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

Figure 7: Start configuration for the sixteen puzzle.

```
        @"SCROLLING=no NAME=enter>\n"
        @"</FRAMESET>\n"
        @"<NOFRAMES>\n"
        @"This software needs a browser "
        @"that can display frames.\n"
        @"</NOFRAMES>\n";
    body = @"";
    bodyargs = @"";
    OutputPageAndExit(mysql, session,
                  title, frames,
                  body, bodyargs);

    // not reached
    [pool release];
    exit(0);
}
```

This concludes the webchat recipe. It is based on a PHP webchat that I wrote some years ago, which in turn was inspired by a script I received from Igor Gilitschenski in July 2000.

# 6 Puzzles

## 6.1 The Sixteen Puzzle

*by Marko Riedel*

### 6.1.1 Idea

The goal is to implement the sixteen puzzle. The classic version of this puzzle consists of fifteen pieces that are placed in a box that is four pieces wide and four pieces high. There is one empty slot. Pieces in the same row as the empty slot can move horizontally and those in the same column can move vertically. The pieces are scrambled and the goal is to restore the start position, which lists the pieces sequentially, starting from the top row. There seems to be some confusion as to whether this puzzle should be called the fifteen puzzle or the sixteen puzzle, we choose the latter. The start position is shown in Fig. 7.

We will let the user choose how many pieces there should be to a single row or column. The user can pick any TIFF image she likes to use as the content of the pieces. These parameters are passed in via the command line.

### 6.1.2 Implementation

There are two components to this application. There is a view that displays the pieces and a controller that creates auxiliary objects like the main menu and the window that holds the view.

Start by declaring some constants: the minimum and maximum number of pieces per row/column and the minimum width and height of a single piece.

```
#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

#define MINDIM 3
```

```
#define MAXDIM 24

#define PIECEMIN 20
```

The class `SixteenView` implements a view that draws all the pieces (a different approach might use one view for every piece). This view has two states: either scrambled or not scrambled. It displays the source image as is when it is in the latter state, otherwise it displays the individual pieces. It must also react to mouse clicks and move pieces accordingly.

Now discuss the variables. The view must remember the state of the board. It stores the state in the two-dimensional array `board`, where the first index determines the row and the second the column. The entries are integers where the integer $m$ represents the piece that sits in row $m/d$, where $d$ is the size, and column $m\%d$ in the source image. The value $-1$ represents the blank. The variable `bdim` is used to store the number of pieces per row or column. We record the position of the blank in the variables `blankX` and `blankY`. There is a flag that indicates whether the view is scrambled or not. We store the source image in the variable `image` and the dimensions of the individual pieces in the variable `pieceSize.` We'll be copying rectangular portions of the source image whose size is given by `pieceSize` in order to construct the image of the scrambled board.

```
@interface SixteenView : NSView
{
    int board[MAXDIM][MAXDIM];
    int bdim;
    int blankX, blankY;

    BOOL scrambled;

    NSImage *image;
    NSSize pieceSize;
}
```

There are only a few methods to `SixteenView`. There is an initializer that is invoked with the origin of the view, the source image, and the number of pieces per row and column.

We implement `acceptsFirstResponder` so that the board can be scrambled with a click on the application's main menu. The method `totalPositioned` tells us how many pieces are in the right position. It will help us determine when the board is properly scrambled and when the user has solved the puzzle.

The method `doMoveX:Y:` is used to move pieces in response to clicks and while we scramble the pieces. Suppose the user clicks on a piece anywhere in the same row or column as the blank. The pieces move so that the blank is in the position that the user clicked on (there is only one way for them to go). The method `scramble:` scrambles the pieces. The last two methods are important: `drawRect:` must draw the view according to its internal state and `mouseDown:` processes mouse clicks by the user.

```
- initAtPoint:(NSPoint)loc withImage:(NSImage *)img
 andDimension:(int)dim;

- (BOOL)acceptsFirstResponder;

- (int)totalPositioned;
- doMoveX:(int)xpos Y:(int)ypos;
- scramble:(id)sender;

- (void)drawRect:(NSRect)aRect;
- (void)mouseDown:(NSEvent *)theEvent;

@end
```

The first thing the initializer does is to compute the frame of the view. It is determined by the position that was passed in as an argument and the size of the image. The width and the height of the image may not always be divisible by the number of rows and columns. Any leftover at the right or at the top will not

be displayed, hence we reduce the frame size by those leftover amounts. We then invoke `NSView`'s method `initWithFrame:` as required to initialize the view.

```
@implementation SixteenView

- initAtPoint:(NSPoint)loc withImage:(NSImage *)img
 andDimension:(int)dim
{
   NSRect frame;
   int x, y;

   frame.origin = loc;
   frame.size = [img size];

   frame.size.width -= ((int)frame.size.width)%dim;
   frame.size.height -= ((int)frame.size.height)%dim;

   [super initWithFrame:frame];
```

The initializer stores the source image in the instance variable and computes the width and height of the individual pieces. It raises an exception if either is too small. We may store the dimension if there was no problem.

```
   image = img;

   pieceSize.width = frame.size.width/dim;
   pieceSize.height = frame.size.height/dim;

   if(pieceSize.width<PIECEMIN ||
     pieceSize.height<PIECEMIN){
      [NSException raise:NSInvalidArgumentException
                  format:@"image_to_small_for_a_%dx%d",
                  dim, dim];
   }

   bdim = dim;
```

Next we indicate that a new `SixteenView` is not in the scrambled state and initialize the board accordingly. The blank goes in the last column of the bottom row.

```
   scrambled = NO;
   for(y=0; y<bdim; y++){
      for(x=0; x<bdim; x++){
         board[y][x] = y*bdim+x;
      }
   }
   board[0][bdim-1] = -1;
   blankX = bdim-1; blankY = 0;

   return self;
}
```

We implement `acceptsFirstResponder` so that the view may respond to action messages.

```
- (BOOL)acceptsFirstResponder
{
   return YES;
}
```

The method `totalPositioned` iterates over all positions on the board and records the number of pieces that are in the correct position.

```
- (int)totalPositioned
{
    int x, y, count = 0;

    for(y=0; y<bdim; y++){
        for(x=0; x<bdim; x++){
            if(board[y][x] == y*bdim+x){
                count++;
            }
        }
    }

    return count;
}
```

The method `doMoveX:Y:` responds to a click at the position that is given by the two arguments. There are four cases. There is nothing to do if the user has clicked on the blank or on a piece that shares neither row nor column with the blank. The interesting cases occur when the user clicks on a piece in the same row or column as the blank. We move the pieces that lie between the click position and the blank, each piece by one position. They move down, if the user clicked above the blank, up, if the user clicked below, right, if the user clicked to the left and finally, left, if the user clicked to the right. These four cases are handled by loops that shift the pieces. We update the position of the blank.

```
- doMoveX:(int)xpos Y:(int)ypos
{
    int mv;

    if(xpos==blankX && ypos==blankY){
        NSBeep();
    }
    else if(xpos==blankX){
        if(ypos>blankY){ // down
            for(mv=blankY; mv<ypos; mv++){
                board[mv][xpos] = board[mv+1][xpos];
            }
        }
        else{ // up
            for(mv=blankY; mv>ypos; mv--){
                board[mv][xpos] = board[mv-1][xpos];
            }
        }
        board[ypos][xpos] = -1; blankY = ypos;
    }
    else if(ypos==blankY){
        if(xpos>blankX){ // left
            for(mv=blankX; mv<xpos; mv++){
                board[ypos][mv] = board[ypos][mv+1];
            }
        }
        else{ // right
            for(mv=blankX; mv>xpos; mv--){
                board[ypos][mv] = board[ypos][mv-1];
            }
        }
```

```
        board[ypos][xpos] = -1; blankX = xpos;
    }
    else{
        NSBeep();
    }

    return self;
}
```

The method `scramble:` scrambles the board. It makes `maxIter` valid moves, where `maxIter` is the total number of pieces and checks that at most a quarter of the pieces are in the correct position. The process repeats if this is not the case (too many pieces placed correctly).

```
- scramble:(id)sender
{
    int maxInPlace = bdim*bdim/4, maxIter = bdim*bdim, i;
```

There is an outer loop that checks the number of correctly placed pieces and an inner loop that carries out `maxIter` moves. Moves are determined as follows: first flip a coin to decide whether to move horizontally or vertically. Then pick a random position that is not equal to the position of the blank and carry out the move. How to avoid the position of the blank? First pick a position that is not the last position, then add one if it is larger than or equal to the blank's position. This guarantees a uniformely random choice between the available positions, which exclude the blank.

```
    do {
        for(i=0; i<maxIter; i++){
            BOOL horizontal = lrand48()%2;
            int mv = lrand48()%(bdim-1);


            if(horizontal==YES){
                if(mv>=blankX){
                    mv++;
                }

                [self doMoveX:mv Y:blankY];
            }
            else{
                if(mv>=blankY){
                    mv++;
                }

                [self doMoveX:blankX Y:mv];
            }
        }
    } while([self totalPositioned]>maxInPlace);
```

The last thing `scramble:` does is to record the fact that the view has been scrambled and mark it for display.

```
    scrambled = YES;
    [self setNeedsDisplay:YES];

    return self;
}
```

The method `drawRect:` is important. It draws the view according to its internal state. Almost everything is done by compositing rectangular portions of the source image. The easy part is first. Simply composite

the rectangle that needs drawing when the view is not scrambled, and draw a boundary around the entire image.

```
- (void)drawRect:(NSRect)aRect
{
   int x, y;

   [[NSColor blueColor] set];

   if(scrambled==NO){
      [image compositeToPoint:aRect.origin
           fromRect:aRect
           operation:NSCompositeCopy];
      [NSBezierPath strokeRect:[self bounds]];
      return;
   }
```

It stays quite simple even when the view is scrambled. Iterate over the positions of the board. The current row and column tell us where content of the piece should go. The actual value of the board at that position tells us where in the source image the piece is to be found. We do not draw the blank.

We compute the source rectangle (where the piece resides in the source image) and the destination rectangle (where it is on the board) for each position of the board, do the composite operation and draw a boundary around the piece; there is nothing to draw for the blank.

```
   for(y=0; y<bdim; y++){
      for(x=0; x<bdim; x++){
         int val = board[y][x],
            px = val%bdim, py=val/bdim;
         NSRect src, dest;

         if(val==-1){
            continue;
         }

         src.origin.x = px*pieceSize.width;
         src.origin.y = py*pieceSize.height;
         src.size = pieceSize;

         dest.origin.x = x*pieceSize.width;
         dest.origin.y = y*pieceSize.height;
         dest.size = pieceSize;


         [image compositeToPoint:dest.origin
              fromRect:src
              operation:NSCompositeCopy];
         [NSBezierPath strokeRect:dest];
      }
   }
}
```

The last method of `SixteenView` is the method `mouseDown:`. It must respond to clicks by the user and reposition pieces accordingly. The first step is to compute the location of the click in the view's coordinate system (not essential here, but good practice). Next we determine what piece has been clicked. There is nothing to do when the user clicks and the view is not scrambled; the same goes for clicks that are not in the same row/column as the blank. If the click makes sense then we process it.

```
- (void)mouseDown:(NSEvent *)theEvent
```

```
{
   NSPoint loc;
   int x, y;

   loc = [theEvent locationInWindow];
   loc = [self convertPoint:loc fromView: nil];

   x = loc.x/pieceSize.width; y = loc.y/pieceSize.height;

   if(scrambled==NO || (x!=blankX && y!=blankY)){
      NSBeep();
      return;
   }

   [self doMoveX:x Y:y];
```

The last step is to check whether the user has solved the puzzle, which occurs when there are as many pieces in place as there are positions on the board, minus the blank. In this case display the complete image right away and congratulate the user, otherwise mark the view as needing display.

```
   if([self totalPositioned]==bdim*bdim-1){
      scrambled = NO;
      [self display];
      NSRunAlertPanel(@"Congratulations!", @"Puzzle solved.",
                  @"Ok", nil, nil);
   }
   else{
      [self setNeedsDisplay:YES];
   }
}

@end
```

The controller is very simple. It builds the window and the view once the application has been launched, sets the main menu and processes command line arguments.

```
@interface Controller : NSObject

- (void)applicationDidFinishLaunching:(NSNotification *)notif;

@end
```

We need to get at the arguments, so we obtain them from the process info object. One of these is the path to the source image, which is a string and will be stored in the variable `imgPath`. There is the variable `img` that holds the image itself. We also obtain the number of pieces per row or column from the command line: this is recorded in the variable `size`. We declare variables to hold the window, the main menu, the `SixteenView` and the content rectangle of the window.

```
@implementation Controller

- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   NSProcessInfo *procInfo = [NSProcessInfo processInfo];
   NSArray *args = [procInfo arguments];

   NSString *imgPath;
   NSImage *img;
   int size;
```

```
SixteenView *sv;

NSMenu *menu = [NSMenu new];

NSWindow *stWin;
NSRect winRect;
```

The first action is to seed the random number generator so that we get different results from a scramble operation every time the program is run. We build the main menu with two entries, one to scramble the view and another to quit the application, and display it right away.

```
srand48(time(NULL));

[menu addItemWithTitle: @"Scramble"
     action:@selector(scramble:)
     keyEquivalent:@""];
[menu addItemWithTitle: @"Quit"
     action:@selector(terminate:)
     keyEquivalent:@"q"];
[NSApp setMainMenu:menu];

[menu display];
```

The arguments are processed next. There must be three of them: the application binary, the size and the image. We raise an exception when we do not have the right number of arguments or when the size is to small for a meaningful game or too large for the board.

```
if([args count]!=3){
   [NSException raise:NSInvalidArgumentException
            format:@"args are: <size> <image>"];
}

size = [[args objectAtIndex:1] intValue];
if(size<MINDIM || size>MAXDIM){
   [NSException raise:NSRangeException
            format:@"size out of range (%d - %d): %d",
            MINDIM, MAXDIM, size];
}
```

We try to read the image from the file and raise an exception if there is a problem.

```
imgPath = [args objectAtIndex:2];
img = [[NSImage alloc] initWithContentsOfFile:imgPath];
if(img==nil){
   [NSException raise:NSInvalidArgumentException
            format:@"no image in %@", imgPath];
}
```

We allocate the instance of `SixteenView` next. The initializer of the view computes the size of the view from the size of the image and we use this value as the size of the window's content rectangle.

```
sv = [[SixteenView alloc]
       initAtPoint:NSMakePoint(0, 0)
       withImage:img andDimension:size];

winRect.origin = NSMakePoint(0, 0);
winRect.size = [sv bounds].size;
```

Finally we allocate the window, set its title, make the `SixteenView` the content view and the initial first responder, center the window and order it to the front.

```
   stWin = [[NSWindow alloc]
            initWithContentRect:winRect
            styleMask:NSTitledWindowMask
            backing:NSBackingStoreBuffered
            defer:NO];
   [stWin setTitle:@"sixteen puzzle"];

   [stWin setContentView:sv];
   [stWin setInitialFirstResponder:sv];

   [stWin center];
   [stWin makeKeyAndOrderFront:nil];
}

@end
```

The function `main` is the same as in the `df` frontend recipe: it allocates the application and the controller and runs the application.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];
   NSApplication *app;

   app = [NSApplication sharedApplication];
   [app setDelegate:[Controller new]];
   [app run];

   [pool release];
   exit(0);
}
```

# 7 Miscellaneous

## 7.1 Color picker with X11

*by Marko Riedel, from a suggestion by Stefan Urbanek*

### 7.1.1 Idea

This recipe is very simple. We build an application with a single window that contains a color well. There are two entries on the application's menu: one to quit the application, and another to pick a color. The cursor changes to a red target cursor when the user clicks the latter, and she may then copy the color of any pixel on the screen to the color well by clicking on the pixel. The color picker code is isolated in a function that is called `GSReadPixelFromScreen.` There is a controller class that acts as a delegate to the application and responds to the action `pick:`.

### 7.1.2 Implementation

Start with the usual include directives. We need the cursor font for the target cursor.

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

#include <X11/Xlib.h>
#include <X11/cursorfont.h>
```

The function `GSReadPixelFromScreen` picks a color in several steps:

- set the cursor to a red target cursor
- grab the pointer
- wait for a button-release event
- release the pointer
- fetch an image containing the pixel that was clicked
- extract and return the color.

The first step is to obtain the current display and its root window.

```
NSColor *GSReadPixelFromScreen()
{
   Display *display = XOpenDisplay(NULL);
   Window root = RootWindow(display, DefaultScreen(display));
```

We prepare the cursor from the font cursor with a target cross shape by setting the color components of its foreground and background colors. The foreground is red (RGB is $(\max, 0, 0)$) and the background white (RGB $(\max, \max, \max)$.) The color components are unsigned, so we get the maximum value by using $-1$ without having to know the actual size of the components.

```
   Cursor cursor = XCreateFontCursor(display, XC_tcross); XColor fgc, bgc;

   fgc.red = -1; fgc.green = 0; fgc.blue = 0;
   bgc.red = -1; bgc.green = -1; bgc.blue = -1;
   XRecolorCursor(display, cursor, &fgc, &bgc);
```

The next step is to grab the pointer. No other application will receive button clicks until we explicitly release the pointer. We must wait until the user chooses a pixel, i.e. until `XNextEvent` returns a button release, in which case we release the pointer. We receive no other events because the event mask argument to `XGrabPointer` only includes button release events.

```
   XGrabPointer(display, root, 0, ButtonReleaseMask,
             GrabModeAsync, GrabModeAsync, None,
             cursor, CurrentTime);

   XEvent event;
   XNextEvent(display, &event);

   XUngrabPointer(display, CurrentTime);
```

The location of the button press in root window coordinates tells us what pixel to fetch. We ask the server for a one pixel image at that location and read the pixel once we have the image. We may then free the image since it is no longer needed.

```
   XImage *ximage =
      XGetImage(display, root,
              event.xbutton.x_root, event.xbutton.y_root,
              1, 1, -1, ZPixmap);
   unsigned long p = XGetPixel(ximage, 0, 0);
   XDestroyImage(ximage);
```

We require the color components rather than the pixel value, so we query the server for the color components of the selected pixel in the default colormap. (We could have extracted the components from the pixel value ourselves e.g. for `TrueColor` visuals, but we choose to keep it simple.)

```
   XColor result; result.pixel = p;
   XQueryColor(display,
            DefaultColormap(display, DefaultScreen(display)),
            &result);
```

This concludes the definition of the color picker function. We have the color components and use them to obtain the appropriate `NSColor` object, taking care to scale the values; $-1$ provides the maximum value for each component when converted to an unsigned short integer.

```
   #define MX ((unsigned short)-1)
   return
      [NSColor colorWithDeviceRed:(float)result.red/MX
             green:(float)result.green/MX
             blue:(float)result.blue/MX
             alpha:1.0];
}
```

It remains to implement the controller. It must assemble the window with the color well after the application finishes launching and pick colors when the user clicks the corresponding menu item.

```
@interface Controller : NSObject
{
   NSColorWell *well;
}

- (void)applicationDidFinishLaunching:(NSNotification *)notif;
- pick:(id)sender;

@end
```

First define the dimensions of the window's content view, allocate the window and initialize it with the title and the minimum size.

```
@implementation Controller

#define WELLDIM 200

- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   NSRect wframe =
      NSMakeRect(0, 0, WELLDIM, WELLDIM);
   NSWindow *window =
      [[NSWindow alloc]
         initWithContentRect:wframe
         styleMask:NSTitledWindowMask | NSResizableWindowMask
         backing:NSBackingStoreBuffered
         defer:NO];
   [window setMinSize:wframe.size];
   [window setTitle:@"Pick"];
```

The well has the same size as the window's content frame and starts out containing the color "blue."

```
   well = [[NSColorWell alloc] initWithFrame:wframe];
   [well setColor:[NSColor blueColor]];
```

We place the well in the view hierarchy by making it the window's content view. We are done with the window, so we center it and place it on screen.

```
    [window setContentView:well];
    [window center];
    [window makeKeyAndOrderFront:self];
}
```

The action `pick` is very simple: it invokes `GSReadPixelFromScreen` and sets the well's color to the color that it returns.

```
- pick:(id)sender
{
    [well setColor:GSReadPixelFromScreen()];
    return self;
}

@end
```

The main function allocates an autorelease pool, the controller and the application object.

```
int main(int argc, char** argv, char **env)
{
    NSAutoreleasePool *pool = [NSAutoreleasePool new];

    Controller *con = [Controller new];

    NSApplication *app = [NSApplication sharedApplication];
```

The menu contains an entry that quits the application and another one that invokes the color picker.

```
    NSMenu *menu = [NSMenu new];
    [menu addItemWithTitle: @"Pick"
          action:@selector(pick:)
          keyEquivalent:@""];
    [menu addItemWithTitle: @"Quit"
          action:@selector(terminate:)
          keyEquivalent:@"q"];
    [NSApp setMainMenu:menu];
```

It remains to connect the controller to the application and start the event loop.

```
    [app setDelegate:con];
    [app run];

    [pool release];
    exit(0);
}
```

## 7.2 Screen grab with X11

*by Marko Riedel, with an idea by Alexander Malmberg*

### 7.2.1 Idea

The goal is to implement a screen grab utility for X11 using GNUstep. This application, or tool rather, should collect data for all windows that are currently visible, be it partially or complete, into a tree that represents the parent-child relationship between them. The most important item per window is the image that it displays. Trees map naturally onto browsers, so we will let the user navigate the window tree with an `NSBrowser`. The application has one window, whose upper portion displays the browser. The lower portion contains a scrollview, whose document view is a special imageview that shows the image in the window at

the time the snapshot was taken (just after launch). We read image data only for the onscreen rectangle of the window. The user can save images to TIFF files. There are two phases to this application. First it queries the X server for the contents of the onscreen windows and builds a window tree; the user views and perhaps saves window contents of the retrieved windows during the second phase.

We'll be working with three classes. The class `XWinData` encapsulates the attributes for an X window and an `NSImage`, which holds the current image obtained for the window. The class `ImageView` is a simple replacement of `NSImageView` and it provides for fast display of an image that is represented by a bitmap and it will be the documentview of the scrollview. The third class is the class `Controller`, which collects the data from the X server, builds the tree of `XWinData` objects starting with the root window and acts as a passive delegate for the browser. It implements a method to save images.

### 7.2.2 `XWinData`

The program starts with the necessary headers, notably the one for `Xlib`.

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>
#include <X11/Xlib.h>
```

The class `XWinData` encapsulates data that describe an X window as well as the image contained in the window. An X window is on a certain display and has a parent. We record these for easy reference. We also record the attributes of the window. We'll be using the location and the size attributes, as well as the map state of the window (whether it is viewable or not). The instance variable `children` makes `XWinData` into a tree structure, with the individual objects being the nodes. We need the tree data to make the set of windows easy to navigate and provide data for the browser.

On of the first things that we'll do is compute the portion of the window that is actually on the screen, in screen coordinates. We only process windows of which at least some part is on the screen. An `XWinData` object also holds a description of itself, which is the name of the window if it has one, and the geometry otherwise. Finally we have the image, which is an `NSImage` object that holds the image that we retrieved for the window.

```
@interface XWinData : NSObject
{
   Display *display;

   Window window, parent;
   XWindowAttributes attribs;

   NSMutableArray *children;

   NSRect onscreen;

   NSString *desc;
   NSImage *image;
}
```

The intializer is straightforward. Note that it doesn't require the array of children or the image, which are computed later in the program, and only if necessary.

```
- initWithDisplay:(Display *)disp
       window:(Window)win
       parent:(Window)par;
```

We need access to the attributes of the window. The methods for this return instance variables. The method `onScreen` indicates whether any portion at all of the window is on the screen.

```
- (NSString *)description;
- (int)map_state;
- (int)xclass;
```

```
- (BOOL)onScreen;
```

The next two methods let us set and retrieve an array of children for the receiver.

```
- setChildren:(NSMutableArray *)cs;
- (NSMutableArray *)children;
```

The entries of the browser's columns should be in alphabetical order so we need a comparison method between two `XWinData` instances that we'll use to sort them.

```
- (NSComparisonResult)cmpdescriptions:(XWinData *)other;
```

If it turns out that some part of a window is on the screen, possibly obscured, then we compute an image from the contents of the window. There is an accessor for this instance variable.

```
- computeImage;
- (NSImage *)image;
```

The last method sends appropriate release messages to the array of children and the description (a string) when the object is being deallocated.

```
- (void)dealloc;

@end
```

We now discuss these methods in detail. The initializer stores its arguments in the appropriate instance variables. It then tries to obtain the window's attributes from the server and raises an exception if it fails.

```
@implementation XWinData

- initWithDisplay:(Display *)disp
        window:(Window)win
        parent:(Window)par
{
   display = disp;
   window = win;
   parent = par;

   children = nil;

   if(!XGetWindowAttributes(display, window, &attribs)){
      NSString *fmt =
         @"couldn't get window attributes: 0x%x";
      [NSException raise:NSGenericException
                format:fmt, (unsigned)window];
   }
```

The next step is to translate the coordinates of the window's origin into screen coordiates (the values in the attribute structure are in the parent's coordinate system).

```
   int tx, ty;
   if(window==attribs.root){
      tx = 0; ty = 0;
   }
   else{
      Window child_return;
      XTranslateCoordinates(display, parent, attribs.root,
                        attribs.x, attribs.y, &tx, &ty,
                        &child_return);
   }
```

We may compute the portion of the window that is on the screen once the coordinates have been translated. We intersect the screen rectangle with the window's rectangle for this purpose. We'll be using the result when we retrieve the image and decide whether the window should be included in the tree.

```
NSRect
   srect =
   NSMakeRect(0, 0,
           attribs.screen->width,
           attribs.screen->height),
   wrect =
   NSMakeRect(tx, ty,
           attribs.width,
           attribs.height);

onscreen = NSIntersectionRect(srect, wrect);
```

Next the initializer builds the description and then it exits. If the name property of the window is set, then we use it as the description. Otherwise we record that there was no name, namely by giving the root window the name `ROOT WINDOW` and other unnamed windows the name {`no title`}. The description of unnamed windows includes the geometry of the window.

```
char *cname;
XFetchName(display, window, &cname);

if(cname!=NULL){
   desc = [NSString stringWithCString:cname];
   XFree(cname);
}
else{
   NSString *fmt = @"%s %dx%d+%d+%d 0x%x";
   desc =
      [NSString stringWithFormat:fmt,
              (window==attribs.root ?
               "ROOT WINDOW" : "{no title}"),
              attribs.width, attribs.height,
              attribs.x, attribs.y,
              (unsigned int)window];
}
[desc retain];
```

The newly initialized object is put into an autorelease pool.

```
   image = nil;

   return [self autorelease];
}
```

The methods `description`, `map_state` and `xclass` return the appropriate instance variables.

```
- (NSString *)description
{
   return desc;
}

- (int)map_state
{
   return attribs.map_state;
}
```

```
- (int)xclass
{
   return attribs.class;
}
```

A window is on the screen if its onscreen-rectangle is not empty.

```
- (BOOL)onScreen
{
   return (NSIsEmptyRect(onscreen) == YES ? NO : YES);
}
```

We need to set and retrieve the array of children. Setting the array will occur at most once, so we don't have to worry about releasing a prior content of the instance variable.

```
- setChildren:(NSMutableArray *)cs
{
   children = cs; [children retain];
   return self;
}

- (NSMutableArray *)children
{
   return children;
}
```

`XWinData` objects are ordered according to the description string for sorting purposes i.e. getting the right order in the browser's columns.

```
- (NSComparisonResult)cmpdescriptions:(XWinData *)other
{
   NSString *odesc = [other description];
   return [desc caseInsensitiveCompare:odesc];
}
```

The next method is perhaps the most important of the entire application. It retrieves the window's image from the window server and stores it in an `NSImage`. We'll be using a bitmap as the image's representation, so we can write data directly into the bitmap's data plane. (We'll use a non-planar bitmap with one plane.)

The first step is to translate the screen coordinates of the onscreen rectangle into window coordinates, so that we can reference the portion of the image that we wish to read. We translate the coordinates from root window coordinates to window coordinates and use the clipped bounds rectangle to obtain the dimensions of the displayed rectangle.

```
- computeImage
{
   int tx, ty;

   if(window==attribs.root){
      tx = 0; ty = 0;
   }
   else{
      Window child_return;
      XTranslateCoordinates(display, attribs.root, window,
                     onscreen.origin.x, onscreen.origin.y,
                     &tx, &ty, &child_return);
   }
   int
      width = onscreen.size.width,
      height = onscreen.size.height;
```

We are now ready to read the image data and invoke `XGetImage` for this purpose. We raise an exception if we couldn't read the image.

```
XImage *ximage =
   XGetImage(display, window,
           tx, ty, width, height,
           -1, ZPixmap);

if(ximage==NULL){
   [NSException
      raise:NSGenericException
      format:@"couldn't get image: 0x%x, %@",
      (unsigned)window, desc];
}
```

We must convert the data in the `XImage` into a bitmap image representation, which we now declare. It has the correct dimensions and uses eight bits per sample (red, green, blue). There is no alpha channel. We retrieve the bitmap's data plane for easy reference. There is only one plane because the bitmap is not planar.

```
NSBitmapImageRep *rep =
   [[NSBitmapImageRep alloc]
      initWithBitmapDataPlanes:NULL
      pixelsWide:width
      pixelsHigh:height
      bitsPerSample:8
      samplesPerPixel:3
      hasAlpha:NO
      isPlanar:NO
      colorSpaceName:NSDeviceRGBColorSpace
      bytesPerRow:width*3
      bitsPerPixel:8*3];

unsigned char *base, *data = [rep bitmapData];
```

We briefly digress to explain how we obtain the colors of the pixels of the image. There is a function `XGetPixel`, which we can use to read the pixels of the image, which are unsigned long integers. If we are not on a `TrueColor` visual, then we obtain the color components for this pixel using the window's color map with a call to `XQueryColor`, otherwise we extract the bits for each color from the pixel's value. Once we have these, it is easy to write them into the bitmap data plane.

We do `TrueColor` visuals first. The visual contains a bit mask for each color component. We must determine where in the pixel value the component starts, and how many bits it takes up. Therefore we declare a "shift" variable (offset, i.e. position) and a "bits" variable (number of bits) for each color component.

```
int x, y;

if(attribs.depth>=8 && attribs.visual->class==TrueColor){
   unsigned long
      rmask = attribs.visual->red_mask,
      gmask = attribs.visual->green_mask,
      bmask = attribs.visual->blue_mask;
   unsigned long
      rshift, rbits, gshift, gbits, bshift, bbits;
```

We apply the following procedure to each mask. First shift the mask to the right as long as the lowest bit is zero. The number of shifts indicates the position of the mask in the pixel. Next shift the mask to the right while the lowest bit is one. The number of these shifts yields the number of bits of the color component. We use at most eight bits. If there are more than eight bits, then we increment the position counter to skip over the least significant bits so that only the eight most significant bits remain.

```
rshift = 0; rbits = 0;
while(!(rmask & 1)){
   rshift++;
   rmask >>= 1;
}
while(rmask & 1){
   rbits++;
   rmask >>= 1;
}
if(rbits>8){
   rshift += rbits-8;
   rbits = 8;
}

gshift = 0; gbits = 0;
while(!(gmask & 1)){
   gshift++;
   gmask >>= 1;
}
while(gmask & 1){
   gbits++;
   gmask >>= 1;
}
if(gbits>8){
   gshift += gbits-8;
   gbits = 8;
}

bshift = 0; bbits = 0;
while(!(bmask & 1)){
   bshift++;
   bmask >>= 1;
}
while(bmask & 1){
   bbits++;
   bmask >>= 1;
}
if(bbits>8){
   bshift += bbits-8;
   bbits = 8;
}
```

We iterate over the entire image and extract the pixel values for each position $(x, y)$.

```
for(y=0; y<height; y++){
   for(x=0; x<width; x++){
      unsigned long c;
      XColor color;

      int centry;

      c = XGetPixel(ximage, x, y);
```

We write the color components directly into the data plane. There are three steps for each component. Shift it to the right so that the component's bit sequence starts at bit zero. Compute a mask that is as long as the number of bits of the component and consists entirely of ones, then extract the value of the

component by computing the bitwise `and` of the component and the mask. Finally, if there were less than eight bits to the component, then pad it with zeros at the right so that it is eight bits long. We output debugging information once the entire image has been processed.

```
            base = data+(y*width+x)*3;

            base[0] = ((c >> rshift) & ((1 << rbits)-1))
                      << (8-rbits);
            base[1] = ((c >> gshift) & ((1 << gbits)-1))
                      << (8-gbits);
            base[2] = ((c >> bshift) & ((1 << bbits)-1))
                      << (8-bbits);
        }
    }

    NSLog(@"%@ TrueColor +%d %d, +%d %d, +%d %d", self,
         rshift, rbits,
         gshift, gbits,
         bshift, bbits);
}
```

We now treat the case when we must use `XQueryColor` to obtain the color. There is only one problem: it is very slow to invoke `XQueryColor` for every pixel. Hence we cache colors locally. The cache consists of an array of pixels and an array of colors. The entries at the same index of these two arrays yield the pixel and the corresponding color. The index for a pixel is the pixel value modulo the cache size. (This is a kind of hash.) We also have an array that indicates whether a particular entry has already been written to. The latter array is initialized so that all entries are marked as empty. You may want to experiment with different sizes of the cache and observe how the performance of the application changes.

```
else{
    Colormap cmap = attribs.colormap;
    #define CSIZE 16384
    unsigned long pixels[CSIZE];
    XColor colors[CSIZE];
    BOOL empty[CSIZE];

    int cind;
    for(cind=0; cind<CSIZE; cind++){
        empty[cind] = YES;
    }
```

We may now iterate over the pixels and write into the bitmap's data plane. Every pixel of the image at some position $(x, y)$ is retrieved.

```
    for(y=0; y<height; y++){
        for(x=0; x<width; x++){
            unsigned long c;
            XColor color;

            int centry;

            c = XGetPixel(ximage, x, y);
```

Next we compute where the color would be in the cache if indeed it has been recorded. Only if we know that the corresponding entry is not empty and its pixel value matches the current pixel may we use the color from the cache. We use `XQueryColor` in all other cases and retrieve the color from the server. The new color is recorded in the cache.

```
            centry = c % CSIZE;

            if(empty[centry]==NO &&
              pixels[centry]==c){
               color = colors[centry];
            }
            else{
               empty[centry] = NO;

               color.pixel = c;
               XQueryColor(display, cmap, &color);

               pixels[centry] = c;
               colors[centry] = color;
            }
```

We now have the color components at the location $(x, y)$ and compute the offset into the data plane. The components are actually sixteen bits, but we work with eight bits per sample and use only the upper eight bits.

```
            base = data+(y*width+x)*3;

            base[0] = color.red >> 8;
            base[1] = color.green >> 8;
            base[2] = color.blue >> 8;
         }
      }

      NSLog(@"%@ color cache", self);
   }
```

The bitmap image representation contains the image at the end of the loop. We are done with the `XImage` and it may be freed. We allocate an image of the right size and make the bitmap its (only) representation. The image is retained and there is a method to access the image.

```
   XDestroyImage(ximage);

   image = [[NSImage alloc]
            initWithSize:NSMakeSize(width, height)];
   [image addRepresentation:rep];
   [image retain];

   return self;
}


- (NSImage *)image
{
   return image;
}
```

The complete functionality of `XWinData` has now been implemented. The method `dealloc` releases the array of children and the image should they have been allocated.

```
- (void)dealloc
{
   if(image!=nil){
      [image release];
```

```
    }
    if(children!=nil){
        [children release];
    }
    [desc release];

    [super dealloc];
}

@end
```

### 7.2.3 ImageView

The class `ImageView` is quite simple. Like `NSImageView`, it stores an image for display. The initializer is the same as in an `NSView`. The image may be recorded and retrieved, and the method `drawRect` does the actual drawing as is standard with views.

```
@interface ImageView : NSView
{
    NSImage *image;
}

- (id)initWithFrame:(NSRect)frameRect;

- (NSImage *)image;
- setImage:(NSImage *)anImage;

- (void)drawRect:(NSRect)aRect;

@end
```

The initializer clears the sole instance variable and invokes the initializer for `NSView.`

```
@implementation ImageView

- (id)initWithFrame:(NSRect)frameRect
{
    image = nil;
    return [super initWithFrame:frameRect];
}
```

Getting and setting the image is easy, but the appropriate `retain/release` messages must be sent.

```
- (NSImage *)image
{
    return image;
}

- setImage:(NSImage *)anImage
{
    if(image!=nil){
        [image release];
    }
    image = anImage;
    if(image!=nil){
        [image retain];
    }
```

```
    [self setNeedsDisplay:YES];
}
```

The actual draw code clips to the rectangle that is being asked for and commands the first representation to draw itself (a bitmap in our case),

```
- (void)drawRect:(NSRect)aRect
{
    // idea for this method by A. Malmberg
    if(image!=nil){
        NSRectClip(aRect);
        [[[image representations]
            objectAtIndex:0] draw];
    }
}

@end
```

### 7.2.4 `Controller` and `main`

The controller object stores the browser and the image view, which is inside a scrollview that does not need to be stored. It also stores the root node of the window tree.

```
@interface Controller : NSObject
{
    NSBrowser *browser;
    ImageView *imgview;
    XWinData *rootData;
}
```

There is an initializer that invokes the method `collectVisibles...` to collect visible on-screen windows before the application object is created and the application launched. This is so that the application's windows do not obscure parts of the screen, whose contents we want to grab.

```
- init;

- (XWinData *)collectVisiblesOnDisplay:(Display *)display
                        window:(Window)window;
```

The controller acts as a passive delegate of the browser and implements the appropriate methods. The key idea is to store the nodes, which are `XWinData` objects, as "represented objects" in the browser's cell. This is the hook that connects the browser to the window tree. There is a method that responds to the browser's cells being selected. It must display the corresponding image.

```
- (int)browser:(NSBrowser *)sender numberOfRowsInColumn:(int)column;
- (void)browser:(NSBrowser *)sender willDisplayCell:(id)cell
        atRow:(int)row column:(int)column;
- (void)selectItem:(id)sender;
```

The controller is also the application's delegate and assembles the main window after the application has been launched.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif;
```

We need a method that will save the currently selected image to a file when the button in the menu is pressed. That button should only be clickable after a window has been selected in the browser, hence the method to validate menu items.

```
- (void)saveAs:(id)sender;

- (BOOL)validateMenuItem:(id <NSMenuItem>)menuItem;

@end
```

The initializer is straightforward: obtain the display and raise an exception if there is a problem. Next obtain the screen and its root window. Then build the window tree.

```
@implementation Controller

- init
{
   Display *display = XOpenDisplay(NULL);
   if(display==NULL){
      [NSException raise:NSGenericException
                  format:@"couldn't_connect_to_display"];
   }

   int screen = DefaultScreen(display);
   Window rootWindow = RootWindow(display, screen);
   rootData =
      [self collectVisiblesOnDisplay:display
           window:rootWindow];

   return self;
}
```

Building the window tree is actually fairly simple. We rely on the procedure `XQueryTree` to walk the tree of X windows and build our tree of `XWinData` objects. The method `collectVisibles...` returns the node that it has created. We start at the root and recursively process children. The first step is to ask for the children of the current window.

```
- (XWinData *)collectVisiblesOnDisplay:(Display *)display
                        window:(Window)window
{
   Window root_return;
   Window parent_return;
   Window *children_return;
   unsigned int nchildren_return;

   XQueryTree(display, window,
           &root_return, &parent_return,
           &children_return, &nchildren_return);
```

The next step is to create the current node. We need the attributes of the window that the initializer of `XWinData` retrieves.

```
   XWinData *current =
      [[XWinData alloc]
         initWithDisplay:display
         window:window
         parent:parent_return];
```

What follows is very important. We identify the windows that go into our tree (not all X windows do). The window must be viewable, it must not be an input-only window, and some part of it must be on screen. If all these conditions are fulfilled, then we record the success in the flag `process` (which tells us whether we should recurse later on) and we compute the image of the window. A window that does not fulfill these

condition causes the corresponding `XWinData` object to be released (recall that we put it into an autorelease pool, so it will be deallocated at some point in the run loop).

```
   BOOL process = NO;
   if([current map_state]==IsViewable &&
      [current xclass]==InputOutput &&
      [current onScreen]==YES){
       process = YES;
       [current computeImage];
   }
   else{
       [current release];
   }
```

The actual recursion follows. There is work to do if the current window was processed and has children. We allocate an array for these children and declare an index that we'll use to iterate over the array `children_return` that we obtained from `XQueryTree`.

```
   if(children_return!=NULL){
      if(process==YES){
         NSMutableArray *children =
            [NSMutableArray arrayWithCapacity:1];
         int ind;
```

The loop iterates over the children and recursively creates a node for each. This node is recorded in the array object if it is not empty.

```
         for(ind=0; ind<nchildren_return; ind++){
            XWinData *child =
               [self collectVisiblesOnDisplay:display
                    window:children_return[ind]];
            if(child!=nil){
               [children addObject:child];
            }
         }
```

We sort the children by their names after all have been obtained. The sorted array is recorded in the instance variable `children` of the current node.

```
         [children sortUsingSelector:@selector(cmpdescriptions:)];
         [current setChildren:children];
      }
```

We must free the list of X windows if indeed there was one. The method returns the newly created node if the window fulfilled our three conditions and `nil` otherwise.

```
      XFree(children_return);
   }

   return (process==YES ? current : nil);
}
```

The controller is the passive delegate of the browser, and the required methods are straightforward. First, we must be able to tell the browser how many rows there are in a given column. This is easy. There is one row that contains the root window in column zero. Otherwise there is a column to the left of the column being loaded, one of its cells is selected, and the new column should display the children of the object represented by the selected cell; hence we ask the selected cell for the represented object and return the number of children of the object.

```
- (int)browser:(NSBrowser *)sender numberOfRowsInColumn:(int)column
{
   if(!column){
      return 1;
   }

   return
      [[[[sender selectedCell] representedObject]
          children] count];
}
```

Second, we must initialize cells that are about to be displayed. A cell should be initialized with data from the represented object, which is the root window in column zero. For non-zero columns the object is the child at position `row` in the array of children of the selected cell, which is located in the previous column.

```
- (void)browser:(NSBrowser *)sender willDisplayCell:(id)cell
        atRow:(int)row column:(int)column
{
   XWinData *represented;

   if(!column){
      represented = rootData;
   }
   else{
      represented =
         [[[[sender selectedCell] representedObject]
             children] objectAtIndex:row];
   }
```

We read the properties of the represented object and record them in the cell. We must know if there are any children; this determines whether the cell is a leaf or not. The string value of the cell is the description string of the object, and the object itself must be recorded in the cell.

```
   int ccount = [[represented children] count];

   [cell setStringValue:[represented description]];
   [cell setRepresentedObject:represented];
   [cell setLeaf:(ccount>0 ? NO : YES)];
}
```

What happens when the user selects a cell in the browser? The method `selectItem` is the action that is invoked in this case. It retrieves the represented object and the image that it contains.

```
- (void)selectItem:(id)sender
{
   XWinData *choice =
      [[(NSBrowser *)sender selectedCell]
         representedObject];
   NSImage *img = [choice image];
```

We place the image in the image view (which is inside a scrollview) and resize the image view to be the same size as the image. This completes our interface with the browser.

```
   [imgview setImage:img];

   NSSize size = [img size];
   NSRect iframe = NSMakeRect(0, 0, size.width, size.height);
   [imgview setFrame:iframe];
```

```
}
```

The remaining code is concerned with the assembly of the main window and the response to save-as-requests. The controller builds the main window after the application has finished launching. Recall that the window consists of an upper part (the browser) and a lower part (the scrollview with the image). Both parts have the same width and different heights. These are given by constants.

```
#define WIDTH_OF_WINDOW  400
#define HEIGHT_OF_BROWSER 150
#define HEIGHT_OF_SCROLLVIEW 350
```

First compute the content rectangle to hold both the upper and the lower part. Then allocate and initialize a window for this size of content view, and set its minimum size and its title.

```
- (void)applicationDidFinishLaunching:(NSNotification *)notif
{
   [[NSApplication sharedApplication] updateWindows];

   NSRect wframe =
      NSMakeRect(0, 0,
               WIDTH_OF_WINDOW,
               HEIGHT_OF_BROWSER+HEIGHT_OF_SCROLLVIEW);
   NSWindow *window =
      [[NSWindow alloc]
         initWithContentRect:wframe
         styleMask:NSTitledWindowMask | NSResizableWindowMask
         backing:NSBackingStoreBuffered
         defer:NO];
   [window setMinSize:wframe.size];
   [window setTitle:@"Grab"];
```

The upper part is next. Compute the frame rectangle of the browser and allocate and initialize it.

```
   NSRect brect =
      NSMakeRect(0, HEIGHT_OF_SCROLLVIEW,
               WIDTH_OF_WINDOW, HEIGHT_OF_BROWSER);
   browser = [[NSBrowser alloc] initWithFrame:brect];
```

The browser should be width-sizable but its height should not change. It is not titled and does not allow multiple selections. There is a minimum value for the width of the individual columns.

```
   [browser
      setAutoresizingMask:
         NSViewWidthSizable | NSViewMinYMargin];
   [browser setTitled:NO];
   [browser setAllowsMultipleSelection:NO];
   [browser setMinColumnWidth:WIDTH_OF_WINDOW/3];
```

Finally we connect the browser to the controller, which is its delegate and implements a passive delegate's methods, as we have seen; the browser's target is the controller and its action the method `selectItem`. We must place it in the view hierarchy.

```
   [browser setDelegate:self];
   [browser setTarget:self];
   [browser setAction:@selector(selectItem:)];

   [[window contentView] addSubview:browser];
```

The lower part contains the scrollview. We compute the frame rectangle and allocate and initialize it.

```
   NSRect srect =
      NSMakeRect(0, 0,
               WIDTH_OF_WINDOW, HEIGHT_OF_SCROLLVIEW);
   NSScrollView *scrollview =
      [[NSScrollView alloc] initWithFrame:srect];
```

The scrollview should be width-sizable and height-sizable and it should have vertical and horizontal scrollers.

```
   [scrollview
      setAutoresizingMask:
         NSViewWidthSizable | NSViewHeightSizable];

   [scrollview setHasHorizontalScroller:YES];
   [scrollview setHasVerticalScroller:YES];
```

The document of the scrollview is an image view of the fast variety that we discussed above. It is initally empty and we choose a frame that will not cause the scrollview to draw scrollers. We set the image view to be the document view of the scrollview.

```
   NSRect irect =
      NSMakeRect(0, 0,
               HEIGHT_OF_SCROLLVIEW/2,
               HEIGHT_OF_SCROLLVIEW/2);
   imgview =
      [[ImageView alloc] initWithFrame:irect];
   [scrollview setDocumentView:imgview];
```

The window is fully assembled once we place the scrollview in the view hierarchy and order the window (now centered) to the front.

```
   [[window contentView] addSubview:scrollview];

   [window center];
   [window makeKeyAndOrderFront:self];
}
```

The penultimate method of the controller lets the user save images to files. The structure is very simple: obtain the selected cell and its image, try to write the bitmap image representation to a TIFF file and signal an error if you fail. We declare `errno` to have access to the systems error messages. We ask for the savepanel and restrict it to TIFF file names. Then we run the panel.

```
extern int errno;

-(void)saveAs:(id)sender
{
   NSSavePanel *savePanel;
   int result;

   savePanel = [NSSavePanel savePanel];
   [savePanel setRequiredFileType:@"tiff"];
   result = [savePanel runModal];
```

We proceed if the user clicked the OK button and ask the panel for the file name, the browser for the currently selected cell, the cell for the object it represents, the object for the image, and the image for its bitmap representation.

```
   if(result == NSOKButton){
      NSString *fname;
```

```
      fname = [savePanel filename];

      XWinData *choice =
         [[browser selectedCell] representedObject];
      NSImage *img = [choice image];
      NSBitmapImageRep *rep =
         [[img representations] objectAtIndex:0];
```

We read the bitmap's TIFF representation into an `NSData` object and try to write this object to a file with the name that the user selected.

```
      NSData *data =
         [rep TIFFRepresentationUsingCompression:
                NSTIFFCompressionPackBits
             factor:0.5];
      BOOL result=[data writeToFile:fname atomically:NO];
```

The remaining code in this method tries to construct a meaningful error message if we couldn't write the file. The message contains the name of the file and the error string for the current value of `errno`, if the latter was set. The message is displayed with an alert panel.

```
      if(result==NO){
         NSString *msg =
            [NSString stringWithFormat:@"Couldn't write %@",
                    fname];

         if(errno){
            char *estr = strerror(errno);
            msg = [msg stringByAppendingFormat:@": %s", estr];
         }

         NSRunAlertPanel(@"Alert", msg, @"Ok", nil, nil);
      }
   }
```

We reset the cursor to an arrow should it still be the ibeam cursor (this used to happen sometimes, not sure if it still does).

```
   [[NSCursor arrowCursor] set];
}
```

The last method of the controller keeps the save-as menu item, whose tag is one, disabled if no window has yet been selected in the browser.

```
- (BOOL)validateMenuItem:(id <NSMenuItem>)menuItem
{
   int row = [browser selectedRowInColumn:0];
   if([menuItem tag]==1 && row==-1){
      return NO;
   }

   return YES;
}

@end
```

This very nearly concludes the discussion of our screen grab application. The routine `main` creates the autorelease pool, the controller and the application. The controller is created before the application so that the screen data do not include the tool's windows, as explained earlier.

```
int main(int argc, char** argv, char **env)
{
   NSAutoreleasePool *pool = [NSAutoreleasePool new];

   Controller *con = [Controller new];

   NSApplication *app;
   app = [NSApplication sharedApplication];
```

The application's menu only contains two entries; one that quits the application and another that lets the user save images.

```
   NSMenu *menu = [NSMenu new];
   [[menu addItemWithTitle: @"Save As"
        action:@selector(saveAs:)
         keyEquivalent:@""]
      setTag:1];
   [menu addItemWithTitle: @"Quit"
        action:@selector(terminate:)
        keyEquivalent:@"q"];
   [NSApp setMainMenu:menu];
```

It remains to connect the controller to the application and run it.

```
   [app setDelegate:con];
   [app run];

   [pool release];
   exit(0);
}
```

This recipe was written with the online Xlib manual kindly provided by Christophe Tronche. Excercise: adapt this recipe to take window borders into account.

## 7.3 Calendar view

*by Marko Riedel*

### 7.3.1 Idea

The purpose of this recipe is to implement a view that displays a calendar. We will attempt to duplicate the look and the functionality of the Unix `cal` command, whose output looks like this:

```
    November 2008
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

What the calendar view actually looks like is shown in figure 8.

The calendar view consists of four major components: a popup button to the left that lets the user choose the month, one to right, that lets the user choose the year, a text view that displays a string that represents the current month and year and a set of seven tables, each one column wide, that display the actual calendar.

The date calculation functions that come with GNUstep in the object `NSCalendarDate` are very powerful. We will see that we can concentrate on designing the GUI.


Figure 8: Calendar view (buttons not part of the view)

### 7.3.2 Calendar view: interface

The program starts with the necessary headers. The popup button that displays the year will offer a range from `MINYEAR` to `MAXYEAR`. We define a string data type that holds a single day. Possible values are: the empty string, or strings like ‘‘`6`’’ or ‘‘`15`’’ up to ‘‘`31`’’.

```
#import <AppKit/AppKit.h>

#define MINYEAR 1998
#define MAXYEAR 2016

typedef char calday[3];
```

The first two instance variables of the view are the two popup buttons for month and year, respectively. The next three store the current state of the calendar: day, month and year. They correspond to the data being displayed. The seven tables (one for each day of the week) are next. We also store the column names (the days of the week) and the names of the twelve months of the year. These two arrays are arrays of strings. The instance variable `monyr` stores the text view that displays the year and the month in the upper center of the view. The last variable holds the actual data for the month being displayed. There are six rows for a maximum of six weeks and seven columns for seven days of the week.

```
@interface Calendar: NSView
{
  NSPopUpButton *monthPopup;
  NSPopUpButton *yearPopup;

  int day;
  int month;
  int year;
```

```
  NSTableView *calTables[7];
  NSArray *calCols;
  NSArray *monthNames;

  NSText *monyr;

  calday calmatrix[6][7];
}
```

The method declarations start with the most important method, the one that computes the calendar and stores the result in `calmatrix`. Next is the standard view initializer that initializes the view and creates the subviews (popup buttons, text view). There are two methods that load data into the table. The first one returns the number of rows, which is always equal to six and the second one reads the appropriate C strings out of the data matrix (`calmatrix`) and returns them as `NSString objects`. The delegate method for selection changes is used to trigger de-selection of the previous selection in the other tables when a cell is selected in a table. There are two action methods for the two popup buttons that are invoked when the year or the month changes and trigger a re-compute of the calendar. Finally we declare `drawRect:`. This method does not do a great deal of drawing, as it is handled mostly by the subviews.

```
- computeCalendar;

- initWithFrame:(NSRect)frame;

- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
- (id)tableView:(NSTableView *)aTableView objectValueForTableColumn:
  (NSTableColumn *)aTableColumn row:(int)rowIndex;
- (void)tableViewSelectionDidChange:(NSNotification *)aNotification;

- (void)yearChanged:(id)sender;
- (void)monthChanged:(id)sender;

- (void)drawRect:(NSRect)aRect;
```

The remaining methods let the user query and set the calendar view. She may query the year, month and day or set a particular date through year, month and day. The view may also be set from a string date, which is parsed by `NSCalendarView` according to one of its formats. Finally, we may select a day of the current month or set the calendar to today's date.

```
- (int)year;
- (int)month;
- (int)day;

- (BOOL)setYear:(int)aYear Month:(int)aMonth Day:(int)aDay;
- (BOOL)setFromString:(NSString *)str format:(NSString *)fmt;

- (BOOL)selectDay;

- today;

@end
```

We need a table view that accepts click-throughs (first mouse), otherwise the user would have to click twice when selecting a day in a different column: once to activate the table and second, to select. We want one click to be sufficient. This is the purpose of `NSTableViewCT`.

```
@interface NSTableViewCT : NSTableView

```

```
- (BOOL)acceptsFirstMouse:(NSEvent *)theEvent;

@end

@implementation NSTableViewCT

- (BOOL)acceptsFirstMouse:(NSEvent *)theEvent
{
  return YES;
}

@end
```

### 7.3.3 Calendar view: implementation

The method `setMonYr` is a "private" method that positions and sets the text field that displays the month and the year in the upper center. To do this, it first creates the string to be displayed (like "August 2005" or "June 2006"), stores it in the text view and sizes the textview to just display this string and nothing else. The x origin of the textview is half the available space after the width of the text view has been subtracted. The y origin is computed from the top. The width of the popups plus some padding is subtracted, as is the width of the text view itself. Finally we move the textview to the new origin and display the entire calendar.

```
@implementation Calendar

- setMonYr
{
  [monyr
   setString:
     [NSString stringWithFormat:@"%@␣%d",
             [monthNames objectAtIndex:(month-1)],
             year]];
  [monyr sizeToFit];

  float
    x = ([self bounds].size.width-[monyr bounds].size.width)/2,
    y = [self bounds].size.height
    - ([yearPopup bounds].size.height+5)
    - [monyr bounds].size.height;

  [monyr setFrameOrigin:NSMakePoint(x, y)];

  [self display];

  return self;
}
```

The method `computeCalendar` is central to the functioning of the view. It starts by creating a calendar date object for the current year and month, which is used to fill the calendar matrix. There are three phases: fill those days of the first week that belong to the previous month with blanks, write the data for the current month, and fill any remaining cells from the next month with blanks. The first `while` loop sets those cells that correspond to the previous month to be empty by writing the string terminator "zero" into the first character.

```
- computeCalendar
{
  int pos = 0;
```

```
 NSCalendarDate *cd =
  [NSCalendarDate
   dateWithYear:year month:month day:1
   hour:0 minute:0 second:0
   timeZone:[NSTimeZone timeZoneWithAbbreviation:@"CEST"]];

 while(pos < [cd dayOfWeek]){
  calmatrix[pos/7][pos%7][0] = 0;
  pos++;
 }
```

The `do` loop writes the actual data for the month. To do this, it writes the current day as a string into the appropriate position of the matrix. It then uses `NSCalendarDate` to advance a day. This is done as long as the latter belongs to the current month. The second `while` loop fills the remaining cells with empty strings, just like the first one. These remaining cells correspond to days of the following month.

```
 int calday = 1;
 do {
  sprintf(calmatrix[pos/7][pos%7], "%d", calday++);
  pos++;

  cd = [cd dateByAddingYears:0 months:0
         days:1 hours:0 minutes:0 seconds:0];
 } while([cd monthOfYear]==month);

 while(pos < 6*7){
  calmatrix[pos/7][pos%7][0] = 0;
  pos++;
 }
```

Once the data matrix has been initialized with the new values, we trigger a reload for every table, i.e. column and deselect all cells of that column. This concludes the core method of the calendar view.

```
 int tb;
 for(tb=0; tb<7; tb++){
  [calTables[tb] reloadData];
  [calTables[tb] deselectAll:self];
 }

 return self;
}
```

The purpose of `initWithFrame:` is mostly to assemble the subviews. It starts by allocating the popup button for the month and sets the position where the menu should appear.

```
- initWithFrame:(NSRect)frame
{
 [super initWithFrame:frame];

 monthPopup =
  [[NSPopUpButton alloc]
   initWithFrame:NSMakeRect(0,0, 100, 20)];
 [monthPopup setPreferredEdge:NSMinYEdge];
```

Next we declare and initialize the array of month names. After that we iterate over all the names and add an item for each month to the popup, setting its tag to the correct numeric value of the month. The target of each item its the calendar view itself and the action is the selector `monthChanged:`. We size the popup to be the right size for the menu of the twelve months.

```
monthNames =
  [NSArray
    arrayWithObjects:
      @"January", @"February", @"March",
    @"April", @"May", @"June", @"July",
    @"August", @"September", @"October",
    @"November", @"December", nil];
RETAIN(monthNames);

int index;

for(index=0; index<12; index++){
  [monthPopup
    addItemWithTitle:
      [monthNames objectAtIndex:index]];

  id item = [monthPopup itemAtIndex:index];
  [item setTag:(index+1)];

  [item setTarget:self];
  [item setAction:@selector(monthChanged:)];
}

[monthPopup sizeToFit];
```

The x origin of the popup is the left margin, i.e. zero. The y origin corresponds to the upper left corner of the view. We set the frame and make the popup a subview of the calendar. We may now proceed to the popup for the year.

```
NSRect mframe = [monthPopup frame];
mframe.origin.x = 0;
mframe.origin.y = frame.size.height−mframe.size.height;

[monthPopup setFrame:mframe];

[self addSubview:monthPopup];
```

We allocate and initialize the popup for the year and set the preferred edge for the menu display.

```
yearPopup =
  [[NSPopUpButton alloc]
    initWithFrame:NSMakeRect(0,0, 100, 20)];
[yearPopup setPreferredEdge:NSMinYEdge];
```

Next we iterate over the years from `MINYEAR` to `MAXYEAR` and add them to the popup, setting the tag of each item to the numeric value of the year it represents. The target of all items is the calendar view and the action is the selector `yearChanged:`. Finally we size it to be just the right size.

```
for(index=MINYEAR; index<=MAXYEAR; index++){
  [yearPopup
    addItemWithTitle:
      [NSString stringWithFormat:@"%d", index]];

  id item = [yearPopup itemAtIndex:(index−MINYEAR)];
  [item setTag:index];

  [item setTarget:self];
```

```
  [item setAction:@selector(yearChanged:)];
}

[yearPopup sizeToFit];
```

It remains to compute the origin of the popup, which should be in the upper right corner. Hence the x origin is obtained by subtracting the width of the popup from the width of the calendar, similarly for the y origin. We set the frame and make the popup a subview.

```
NSRect yframe = [yearPopup frame];
yframe.origin.x = frame.size.width−yframe.size.width;
yframe.origin.y = frame.size.height−yframe.size.height;

[yearPopup setFrame:yframe];

[self addSubview:yearPopup];
```

The assembly of the two popups is completed by setting them to the current month and the current year as provided by `NSCalendarDate`.

```
NSCalendarDate ∗cd =
  [NSCalendarDate calendarDate];

month = [cd monthOfYear];
[monthPopup
  selectItemAtIndex:month−1];

year = [cd yearOfCommonEra];
[yearPopup
  selectItemAtIndex:year−MINYEAR];
```

The next section is concerned with the seven one-column tables that show the actual calendar. We start by computing the width of a single column (and hence, table). There are seven columns, one for each day of the week. Next we initialize the array of column names.

```
int width = (frame.size.width−2)/7;

NSRect cframe =
  NSMakeRect(1, 1,width,
           frame.size.height∗0.8);

calCols =
  [NSArray arrayWithObjects:@"Sun",
         @"Mon", @"Tue", @"Wed", @"Thu",
         @"Fri", @"Sat", nil];
RETAIN(calCols);
```

We prepare an enumerator so that we may iterate over the column names. The first task of each iteration is to create a click-through table and store it in the `calTables` instance variable. Its data source is the calendar itself, and so is its delegate. We do not allow column selection.

```
NSEnumerator ∗colEnum = [calCols objectEnumerator];
NSString ∗colName; int ind = 0;

while((colName = [colEnum nextObject]) != nil){
  NSTableView ∗calTable =
    [[NSTableViewCT alloc] initWithFrame:cframe];

  calTables[ind] = calTable;
```

```
  [calTable setDataSource:self];
  [calTable setDelegate:self];
  [calTable setAllowsColumnSelection:NO];
```

The second task is to create the single column that fills every table. We create the column, set it to be not editable, set its header cell to be the name of the weekday and its width to be a seventh of the available space. We add it to the table, shift the table to the right to its proper position and make it a subview.

```
  NSTableColumn *column =
    [(NSTableColumn *)[NSTableColumn alloc]
                 initWithIdentifier:colName];
  [column setEditable:NO];
  [[column headerCell] setStringValue:colName];
  [column setWidth:(frame.size.width-2)/7];

  [calTable addTableColumn:column];

  [calTable
    setFrameOrigin:
      NSMakePoint(1 + ind*width, 1)];
  [self addSubview:calTable];
```

The assembly is completed by shifting the header view of the table upwards and to the left to its proper position. We get its frame, and shift the x origin to the left and the y origin upwards. We set the frame of the header and add it as a subview.

```
  NSTableHeaderView *calHeader = [calTable headerView];

  NSRect hframe = [calHeader frame];
  hframe.origin.x = ind*width;
  hframe.origin.y += [calTable frame].size.height;
  [calHeader setFrame:hframe];

  [self addSubview:calHeader];

  ind++;
}
```

The last task is to create the text view for the month and the year in the upper center region. We create it and set it to be neither editable nor selectable. It is resizable, as it changes size when the parameters change. It does not draw its background. We set an appropriate font, initialize it with today's date, which we read earlier, and make it a subview.

```
monyr =
  [[NSText alloc]
    initWithFrame:NSMakeRect(0, 0, 10, 10)];
[monyr setEditable:NO];
[monyr setSelectable:NO];

[monyr setVerticallyResizable:YES];
[monyr setHorizontallyResizable:YES];

[monyr setDrawsBackground:NO];

[monyr setFont:[NSFont labelFontOfSize:24]];

[self setMonYr];
[self addSubview:monyr];
```

This concludes the initializer. We set the day to today's date and return.

```
  day = [cd dayOfMonth];
  [self selectDay];

  return self;
}
```

The next three methods are concerned with managing the data for the seven table views. All tables have six rows.

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
  return 6;
}
```

We must be able to provide an object value for a given row and column when asked for it. To do this, we iterate over the column tables until we find the right table (the one being queried). We read the value for the appropriate row out of the `calmatrix` instance variable and convert it into an Objective C string object. This is the required datum, which we return.

```
- (id)tableView:(NSTableView *)aTableView objectValueForTableColumn:
  (NSTableColumn *)aTableColumn row:(int)rowIndex
{
  int col = 0;

  for(col=0; col<7; col++){
    if(calTables[col] == aTableView){
      break;
    }
  }

  // NSLog(@"%d %d %s", rowIndex, col, calmatrix[rowIndex][col]);
  return [NSString stringWithFormat:@"%s", calmatrix[rowIndex][col]];
}
```

We must coordinate the selection among the seven tables. When the user clicks on a cell in some column, we must de-select the previously selected cell. Therefore we implement the delegate method "selection did change". It is invoked at two levels: first, when a new date is selected in a table, and second, recursively, when the remaining tables resign their selection on being required to do so in a loop from the first level. This second case starts the method. If the selection has been resigned (second level), then there is nothing more to be done.

```
- (void)tableViewSelectionDidChange:(NSNotification *)aNotification
{
  NSTableView *tb = [aNotification object];

  int row = [tb selectedRow];
  if(row==-1){
    return;
  }
```

The first level is a loop that iterates over the seven column tables. If a table is not the one that has newly been selected, then it is asked to resign its selection, if any (which triggers the recursive call on level two). Otherwise, if the user has not clicked an empty cell, that cell's value becomes the new day value of the calendar.

```
  day = -1;
```

```
  int col;
  for(col=0; col<7; col++){
    if(tb != calTables[col]){
      [calTables[col] deselectAll:self];
    }
    else{
      if(strcmp(calmatrix[row][col], ""){
        day = atoi(calmatrix[row][col]);
      }
    }
  }
}
```

The action method `yearChanged` reads the new year from the tag of the sender, sets the day to be void and recomputes the calendar.

```
- (void)yearChanged:(id)sender
{
  year = [sender tag];
  [self setMonYr];

  day = -1;
  [self computeCalendar];
}
```

The action method `monthChanged` reads the new month from the tag of the sender, sets the day to be void and recomputes the calendar.

```
- (void)monthChanged:(id)sender
{
  month = [sender tag];
  [self setMonYr];

  day = -1;
  [self computeCalendar];
}
```

The method `drawRect:` draws a tiled rectangle as the boundary of the view. The rest of the drawing is handled by the subviews.

```
- (void)drawRect:(NSRect)aRect;
{
  NSRectEdge sides[] = {
    NSMinXEdge, NSMaxXEdge, NSMinYEdge, NSMaxYEdge};
  float grays[] = {
    NSWhite, NSBlack, NSBlack, NSWhite};

  NSDrawTiledRects([self bounds], [self bounds], sides, grays, 4);
}
```

There are three simple accessors, one for the year, another for the month, and a third for the day.

```
- (int)year
{
  return year;
}

- (int)month
{
```

```
  return month;
}

- (int)day
{
  return day;
}
```

The setter method `setYear:Month:Day:` sets the calendar to a given date. It checks first whether the year and the month hold acceptable values.

```
- (BOOL)setYear:(int)aYear Month:(int)aMonth Day:(int)aDay;
{
  if(aYear<MINYEAR || aYear>MAXYEAR){
    return NO;
  }

  if(month<1 || month>12){
    return NO;
  }
```

Next it tries to re-compute the calendar, while remembering the previous values, so the change may be undone if necessary. If the given day cannot be selected for the new month/year pair, the calendar reverts to the current date.

```
  int prevYear = year, prevMonth = month, prevDay = day;

  year = aYear; month = aMonth; day = aDay;
  [self computeCalendar];

  if([self selectDay]==NO){
    year = prevYear;
    month = prevMonth;
    day  = prevDay;

    [self computeCalendar];
    [self selectDay];

    return NO;
  }
```

Otherwise the change has been successful and we update the two popups to reflect the new data, as well as the text field in the upper center.

```
  [monthPopup
    selectItemAtIndex:month-1];

  [yearPopup
    selectItemAtIndex:year-MINYEAR];

  [self setMonYr];

  return YES;
}
```

There is a second setter method, namely `setFromString:format:`, which attempts to compute a valid calendar date corresponding to the string and the format. If this fails, the setter does nothing. Otherwise it extracts year, month and day and invokes `setYear:Month:Day:`.

```
- (BOOL)setFromString:(NSString *)str format:(NSString *)fmt;
{
  NSCalendarDate *cd =
    [NSCalendarDate
      dateWithString:str calendarFormat:fmt];

  if(cd==nil){
    return NO;
  }

  return
    [self
      setYear:[cd yearOfCommonEra]
      Month:[cd monthOfYear]
      Day:[cd dayOfMonth]];
}
```

The method `selectDay` attempts to select the right cell in the appropriate table corresponding to the current value of the instance variable `day`. To do this, it iterates over all positions, sweeping from left to right and top to down. If it finds the string corresponding to the current date, it selects the corresponding cell in the its table. Comparisons are done via strings and `dstr` holds a string representing the value of `day`.

```
- (BOOL)selectDay
{
  char dstr[4]; sprintf(dstr, "%d", day);
  int pos;
  for(pos=0; pos<6*7; pos++){
    if(!strcmp(calmatrix[pos/7][pos%7], dstr)){
      [calTables[(pos%7)]
              selectRow:(pos/7)
              byExtendingSelection:NO];
      return YES;
    }
  }

  return NO;
}
```

The implementation of the calendar view concludes with the method `today`, which sets it to the current date. It computes the current date and invokes `setYear:Month:Day:` with the corresponding arguments.

```
- today
{
  NSCalendarDate *cd =
    [NSCalendarDate calendarDate];

  [self
    setYear:[cd yearOfCommonEra]
    Month:[cd monthOfYear]
    Day:[cd dayOfMonth]];

  return self;
}

@end
```

The remainder of the recipe consists in basic tasks that set up a test application for the calendar view. Start by defining the width and height of the calendar view. Create an autorelease pool, an application and a trivial menu with a quit button.

```
#define CAL_WIDTH 300
#define CAL_HEIGHT 180

int main(int argc, char** argv, char **env)
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];

  NSApplication *app;
  app = [NSApplication sharedApplication];

  NSMenu *menu = [NSMenu new];
  [menu addItemWithTitle: @"Quit"
       action:@selector(terminate:)
       keyEquivalent:@"q"];
  [NSApp setMainMenu:menu];
```

Next create a window for the calendar with the right dimensions. Set its title to be "calendar". Create the calendar and attach it to the window. Center the window and order it to the front.

```
  NSWindow *calwin =
    [[NSWindow alloc]
      initWithContentRect:
        NSMakeRect(100, 100, CAL_WIDTH, CAL_HEIGHT)
      styleMask:NSTitledWindowMask
      backing:NSBackingStoreRetained
      defer:NO];
  [calwin setTitle:@"Calendar"];

  Calendar *cal =
    [[Calendar alloc]
      initWithFrame:
        NSMakeRect(0, 0, CAL_WIDTH, CAL_HEIGHT)];

  [calwin setContentView:cal];
  [calwin center];
  [calwin orderFront:nil];
```

Last, run the application and when it is done, release the autorelease pool and exit.

```
  [app run];

  [pool release];
  exit(0);
}
```