CSCI 2132: Software Development

Lab 7: C Basic Types and Formatted I/O

Synopsis

In this lab, you will:

- Learn more about C's types
- Practice working with `printf` and `scanf`
- Learn to work with the command line arguments of C programs

Contents

Overview

In this lab, you will implement the Unix `cal` tool, which prints a calendar view of a single month or entire year to `stdout`. In order to do this, you have to learn to work with command line arguments and character arrays, and you have to review your knowledge of the `scanf` and `printf` commands. Before starting on this journey, you will briefly explore the representations of various types in memory and use character and integer arithmetic to print the binary representation of an integer to `stdout`.

Step 1: Login and lab setup

Log into your `bluenose` account and create a directory `lab7` under your directory `~/csci2132/svn/CSID` and add it to SVN.

Step 2: The size of basic data types

Write a program that prints the sizes of the following types to `stdout`:

- `char`
- `bool`
- `int`
- `short int`
- `unsigned long int`
- `float`
- `double`
- `long double`

A sample program that prints the size of `long double` looks like this:

type_sizes.c

```
#include <stdio.h>

int main() {
  printf("sizeof(long double) = %u\n", sizeof(long double));
  return 0;
}
```

Complete this program so it prints the sizes of all the types above, run it, and verify that these sizes match what you expected based on what we discussed in class.

You will get an error message for the `bool` type. Even in C99, the `bool` type is not built into the compiler itself; it needs to be made available with an include file. Add `#include <stdbool.h>` to your file. Now it should compile fine.

Step 3: Help from the compiler with avoiding mistakes

There is, in fact, something wrong with `type_sizes.c` even though your compiler doesn't tell you so. The `-Wall` option of `gcc` turns on all warnings, even the most nitpicky ones. (The general format of this option is `-W` followed by a list of warnings that should be turned on; `all` turns on all warnings.)

Compile the above code *with* the `-Wall` option:

```
$ gcc -o type_sizes -Wall type_sizes.c
type_sizes.c: In function ‘main’:
type_sizes.c:5:3: warning: format ‘%u’ expects argument of type ‘unsigned int’,
but argument 2 has type ‘long unsigned int’ [-Wformat=]
   printf("sizeof(char) = %u\n", sizeof(char));
   ^
...
```

A potential problem the compiler has identified here is that you ask `printf` to print the result of `sizeof(char)` as an unsigned integer but this value is in fact an unsigned *long* integer. Consult the `printf` manpage if necessary to find out which conversion specifier needs to be used to print unsigned long integers. Correct all conversion specifiers so that `gcc -o type_sizes -Wall type_sizes.c` finishes without warnings.

A note on the `-Wall` option: It can feel a little annoying to be told even about mostly harmless transgressions of the code you write. It is still a good idea to *always* use the `-Wall` option because what may seem like a harmless transgression now may be less harmless later.

Step 4: The binary representation of integers

In class, we discussed the binary representations of integers in C. Unfortunately, `printf` does not have a conversion specifier that shows this binary representation. No problem. We'll roll our own then and, for good measure, increase the readability of the output by breaking it up into 4-bit blocks. To keep things simple, you will only deal with 16-bit numbers, for now.

As part of this exercise, we'll work with functions and pointers, something we have not discussed in class yet.

Here is our main function:

int_rep.c

```
int main() {
  short int x;
  printf("Enter a number: ");
  scanf("%hd", &x);
  printf("The binary representation is %s\n", binary_rep(x));
  return 0;
}
```

This function should hold little mystery to you by now: It prompts the user for a number, reads it using `scanf`, and then prints its binary representation, which is a string produced by the `binary_rep` function. The only novelty is the conversion specifier `%hd`. Remember that `%d` expects an integer. `%ld` reads a long integer. `%hd` reads a short integer (think about it as "half width").

If you were to implement the `binary_rep` function in Java, you could place it anywhere in your code (apart from the fact that it would have to be a method of a class, not just a standalone function). In C, a function must be declared before it is used for the first time. So you need to place the following implementation of the `binary_rep` function *before* the `main` function:

int_rep.c

```
char *binary_rep(short int x) {
  static char bits[20];
  char *bit = bits + 19;
  unsigned char i, j;
  for (i = 0; i < 4; ++i) {
    *(bit--) = ' ';
    for (j = 0; j < 4; ++j) {
      *(bit--) = '0' + (x & 1);
      x >>= 1;
    }
  }
  bits[19] = 0;
  return bits;
}
```

Now, this function contains a whole list of interesting little nuggets:

String representation: Strings in C are represeted as arrays of characters (`char bits[20]` in this case). C strings do not store their lengths anywhere; the end of a string is marked by a `NUL` character (`'\0'`). Here, you construct a string representation of a number consisting of four 4-bit blocks separated by spaces. Thus, characters 0–3, 5–8, 10–13, and 15–18 are digits, characters 4, 9, and 14 are spaces and, to mark the end of the string, character 19 is set to 0 in the second-last line of the function. One may argue that this line should read `bits[19] = '\0'`. This function uses the dual personality of characters here and simply assigns the 8-bit integer 0 to `bits[19]`.

Local static variables: Normally, variables defined in a function (here, `bits`, `bit`, `i`, and `j`) exist only while the function is active; they disappear after the function returns. By defining a local variable as `static` (here, `static char bits[20]`), you ensure that

- The variable continues to exist beyond the return of the function.
- Multiple calls to the function use the same memory location(s) for this variable. (Most of the time, this is a bad thing, especially in multi-threaded code, but we do not need to worry about this here.)

(Effectively, this is the same as a static variable defined outside of any function, but the variable can be accessed only inside the function even though its lifetime is not limited to the invocation of the function.)

Pointers and pointer arithmetic: We already discussed the `&` operator is class, which returns the address of a variable. In C lingo, the result of the `&` operator is a *pointer to* the variable. The "inverse" operation is dereferencing a pointer using `*`. So, if `x` is a pointer to an integer value, then `*x` is the integer value stored at the memory location addressed by the pointer. Using `*x` on the left-hand side of an assignment operation allows us to assign a new value to the memory location referenced by `x`.

Now, C treats a variable holding an array as a pointer to the first array element. In the third line of the function, `bits` is used in this fashion, as the address of the first element in the `bits[20]` array. This line defines a variable `bit` that stores a pointer to a `char` value.

This variable is initialized to point to, not `bits` but `bits + 19`. This is another interesting feature of C: you can do simple arithmetic with pointers. Given a pointer `p` and an integer x, `p + x` is a pointer to the memory location that is `x` "positions" after `p`. But what's a position? That depends on the type of the pointer. If the pointer type is "pointer to `char`", as is the case for the `bits` pointer, then the width of the element referred to by `bits` is `sizeof(char)`, which is 1. Thus, one "position" is just one byte. `bits + 19` thus refers to the address 19 bytes after the address identified by `bits`. Therefore, the `bit` pointer is initialized to point to the 19th character (counting from 0) in the `bits` array. If `bits` were an array of `int`s, then every element in this array would be 4 bytes wide. In that case, one "position" would be 4 bytes and `bits + 19` would refer to the address 76 bytes after the address identified by `bits`. Clearly, this is useful: if `arr` is an array of elements of any type, then `arr + i` is nothing but the address of the ith element in `arr`.

Given that `*` dereferences a pointer and you can do pointer arithmetic, there's a different way to access the `i`th element in an array: Normally, you would write `arr[i]` to access the `i`th element in `arr`. You can write `arr + i` to obtain a pointer to the `i`th element in `arr`; and then you can dereference this pointer to obtain access to this element: `*(arr + i)`. There is absolutely no difference in C between `arr[i]` and `*(arr + i)`. Even if `arr` is not the name of an array (as is the case for `bits` here) but an arbitrary pointer to some data item, you can still write `*(arr + i)` or `arr[i]` to refer to the element `i` positions after the address referenced by `arr`. I generally use array indexing when I index into an array and pointer arithmetic when working with plain pointers, but this is purely a stylistic choice.

8-bit integers: The outer loop in the function iterates over the 4-bit blocks in the output. The inner loop iterates over the bits in the current block. Since both loop variables take on integer values between 0 and 4, they fit in a single byte, so they can be represented as `unsigned chars`, which is a perfectly valid type to represent small integer values. (On current hardware, it is possible that this is not advisable because many computers work more efficiently with values that match their word size; even working with smaller values may be less efficient. I made this choice here only to demonstrate that this can be done.)

Prefix and postfix increment: C has two increment (`++`) and two decrement (`--`) operators. The side-effect of both operators is the same: the variable is increased or decreased by one. The *value* of the expression differs based on the position of the operator. The expression `x++` (postfix increment) increases `x` by one but has as its value the value of `x` before incrementing it. The expression `++x` (prefix increment) increases `x` by one and has at its value the new value of `x`. The behaviour of prefix and postfix decrement is analogous.

The details of implementing these operators at the machine code level imply that postfix increment and decrement are less efficient than prefix increment and decrement. Thus, you should prefer to use the prefix version whenever possible. Here, you use prefix increment to increment `i` and `j` in the for-loops, because the *value* of the expression does not matter at all; the side-effect of incrementing `i` or `j` is what matters.

On the other hand, the two assignments on the first lines of the loops use postfix decrement. Since the value of the expression `bit--` is the same as that of the expression `bit`, `' '` or `'0' + (x & 1)` is thus assigned to the memory location currently referenced by `bit`. After the operation, however, `bit` refers to the position one position to the left of the position currently assigned because `bit--` decreases `bit` by 1.

Mixed character and integer arithmetic: The inner for-loop uses the line `*(bit--) = '0' + (x & 1)`. The expression `x & 1` is 0 if the rightmost bit of `x` is 0, and 1 if the rightmost bit of `x` is 1. Whatever this value is, it is added to `'0'` to obtain either the character `'0'` itself (if `x & 1` is 0) or the character `'1'` (the character after `'0'` if `x & 1` is 1). This type of arithmetic works painlessly because character constants such as `'0'` and integer expressions such as `x & 1` can be mixed freely in arithmetic expressions.

The final code looks like this:

int_rep.c

```
#include <stdio.h>

char *binary_rep(short int x) {
  static char bits[20];
  char *bit = bits + 19;
  unsigned char i, j;
  for (i = 0; i < 4; ++i) {
    *(bit--) = ' ';
    for (j = 0; j < 4; ++j) {
      *(bit--) = '0' + (x & 1);
      x >>= 1;
    }
  }
  bits[19] = 0;
  return bits;
}

int main() {
  short int x;
  printf("Enter a number: ");
  scanf("%hd", &x);
  printf("The binary representation is %s\n", binary_rep(x));
  return 0;
}
```

Play with this code to

- Inspect the binary representations of various positive and negative integer values and check that they match your understanding.
- Understand exactly how this code works. (You should be able to explain every line to anyone who asks you about this. If you do not understand what a given line does, add `printf` statements to inspect the values of variables as the code runs.)

Now modify the code so that the integer it reads and converts into a binary representation has 32 bits (`int` as opposed to `short int`). You need to change variable types, the number of iterations of the outer loop in the `binary_rep` function, and the size of the `bits` array. To determine the required size, calculate how many 4-bit blocks there are in a 32-bit number and recall that each such block is succeeded by a space or `NUL` character.

Step 5: How to pick the compiler's C standard

The variables `i` and `j` in the `binary_rep` function are used only inside the loop. In Java, this would be a case when you would move the declaration of these variables into the loop, like this:

int_rep.c

```
char *binary_rep(short int x) {
  static char bits[20];
  char *bit = bits + 19;
  for (int i = 0; i < 4; ++i) {
    *(bit--) = ' ';
    for (int j = 0; j < 4; ++j) {
      *(bit--) = '0' + (x & 1);
      x >>= 1;
    }
  }
  bits[19] = 0;
  return bits;
}
```

Change the file `int_rep.c` so the `binary_rep` function looks like above. Now try to compile it:

```
$ gcc -Wall -o int_rep int_rep.c
int_rep.c: In function ‘binary_rep’:
int_rep.c:6:3: error: ‘for’ loop initial declarations are only allowed in C99 mode
   for (unsigned char i = 0; i < 4; ++i) {
   ^
int_rep.c:6:3: note: use option -std=c99 or -std=gnu99 to compile your code
int_rep.c:8:5: error: ‘for’ loop initial declarations are only allowed in C99 mode
     for (unsigned char j = 0; j < 4; ++j) {
     ^
```

This didn't go well, but the compiler helpfully tells you what's wrong: you used a C99 feature (loop variable declaration as part of the loop) that is not supported in gcc's default C89 mode. The `-std` option of gcc allows you to choose the C standard you want to use when compiling your code. Here, you should choose the C99 or C11 standard:

```
$ gcc -Wall -std=c99 -o int_rep int_rep.c
```

This should not produce any errors or warnings and you can run your code again.

Step 6: How to read command line arguments

As programs written in any other programming language, C programs have access to their command line arguments. The way this is done in C is somewhat akin to the approach taken in Java: the command line arguments are passed as arguments to the call of the `main` function. A difficulty to work around is that arrays in C do not know their size; they are simply indexable chunks of memory. To let you know how many command line arguments there are, the `main` function takes two arguments: the number, `argc`, of command line arguments and an array, `argv`, of strings storing the command line arguments. This leads to the following definition of the `main` function:

```
int main(int argc, char **argv) {
  ...
}
```

The argument count, `argc`, includes the name of the program itself. That is, if the program is invoked without arguments, `argc == 1`; if the program is invoked with one argument, `argc == 2`; and so on.

The second argument probably looks weird to you: it says that `argv` is a pointer to a pointer to `char`. Recall that arrays are represented by pointing to the first element in the array. Thus, this says that `argv` is an array of pointers to `char`. Each pointer in this array points to the first character of a command line argument; each command line argument is the string starting at the identified character position and, as all strings in C, ends at the first `NUL` character encountered.

To try out how command line arguments are passed to C programs, implement the following simple piece of code and observe its output for different sets of command line arguments you provide to the program:

cmdline.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Number of arguments = %d\n", argc);
  for (int i = 0; i < argc; ++i) {
    printf("Argument %d: %s\n", i, argv[i]);
  }
  return 0;
}
```

```
$ gcc -Wall -std=c99 -o cmdline cmdline.c
$ ./cmdline
Number of arguments = 1
Argument 0: ./cmdline
$ ./cmdline abc 3 "an argument" -x
Number of arguments = 5
Argument 0: ./cmdline
Argument 1: abc
Argument 2: 3
Argument 3: an argument
Argument 4: -x
```

Reimplementing Unix's `cal` tool

Unix's `cal` tool prints a simple calendar view to `stdout`. Log into `bluenose` and try it out. Try `cal`, `cal 1 2019`, and `cal 2019`. They show a calendar view of the current month, a calendar view of January 2019, and a view of the entire year 2019.

`cal` accepts a few command line arguments that customize its behaviour. We do not care about these here. We want to implement our own tool, `mycal`, that accepts 0–2 arguments. Without argument, it prints a view of the current month. With one argument, this argument is interpreted to be the year to be displayed. With two arguments, the first argument must be a month value between 1 and 12, and the second argument is the year; in this case, a display of the chosen month in the chosen year should be the output.

In order to avoid difficulties with dealing with leap years and to allow you to assume that the year has exactly 4 digits, the `mycal` tool accepts only years between 1900 and 9999. (Leap year calculations are fairly predictable also for earlier years, but even Unix's `cal` tool erroneously marks the year 4 as a leap year and 3 as a non-leap year, even though the opposite was the case.)

Step 7: Analyze the command line arguments of the calendar tool

First you need to parse the command line arguments provided by the user. You need to store the provided month in a variable `month` and the provided year in a variable `year`. Absent arguments should be represented using negative values for now. This leads to the following first skeleton of the `mycal` tool:

mycal.c

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

void usage(char *progname) {
  printf("USAGE: %s [month] [year]\n", progname);
  exit(1);
}

int main(int argc, char **argv) {
  int month, year, nchars;

  switch (argc) {
    case 1:  month = year = -1;
             break;
    case 2:  month = -1;
             if (sscanf(argv[1], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    case 3:  if (sscanf(argv[1], "%d%n", &month, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || month < 1 || month > 12) {
               usage(argv[0]);
             }
             if (sscanf(argv[2], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[2]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    default: usage(argv[0]);
  }
  printf("Month = %d\n", month);
  printf("Year = %d\n", year);
  return 0;
}
```

Implement this and play with it to verify that it works correctly. In particular, you should verify that you get a usage message for incorrect arguments, including arguments that are not numbers and arguments outside the permissible range for months or years.

The logic is as follows:

- If there are no command line arguments (`argc == 1`; recall that the program name itself is always the first entry in `argv`), then the absence of arguments is recorded by setting `month` and `year` to −1.
- If there is one argument (`argc == 2`), then this argument should be a valid year. You set `month = -1` and then use the `sscanf` function to try to convert the first command line argument `argv[1]` to an integer.

  The `sscanf` function behaves exactly like `scanf`, except that its input is not `stdin` but the string provided as the first argument. You check `sscanf`'s return value to ensure that it successfully parsed an integer from `argv[1]`. To ensure that there are no trailing characters that couldn't be parsed, you use the `%n` conversion specifier to assign the number of characters read to a variable `nchars`. Then you compare `nchars` to the length of the whole command line argument `argv[1]`, which you compute using the function `strlen` provided in the `string.h` header file. If the number of characters read and thus stored in `nchars` is less than `strlen(argv[1])`, then there were junk characters that could not be parsed as part of the integer. Finally, you check that the given year is between 1900 and 9999. If any of these conditions fails, you print a usage message and exit.
- If there are two arguments (`argc == 3`), the approach is similar. You parse the first argument as a month and check that it is between 1 and 12, and the second argument as a year.
- Finally, if there are more than two arguments, the `default` branch, then this is not a valid use of the program and you print a usage message.

Step 8: Find the weekday of the first day of a given month

To format a calendar view of a given month, you need to map days of the month to weekdays. This is easy once you know the first weekday of the month. The first function you need to implement to do this is

```
int find_first_day(int month, int year) {
  ...
}
```

The argument is a valid month and a valid year. The return value is a number between 0 and 6, where 0 = Sunday and 6 = Saturday.

The idea is simple: you figure out that January 1, 1900 was a Monday. Then you express the given date as the number of days since January 1, 1900 and take the remainder of division by 7 to calculate the weekday of the given date.

If the current year is y, then $y' = y - 1900$ full years have passed since January 1, 1900. The number of days is

$$365y' + \lfloor (y'-1)/4 \rfloor - \lfloor (y'-1)/100 \rfloor + \lfloor (y'+299)/400 \rfloor$$

because every 4th year is a leap year, except if it is divisible by 100 and not divisible by 400.

Since $365 \bmod 7 = 1$, the weekday of January 1 in year y is therefore

$$(y' + \lfloor (y'-1)/4 \rfloor - \lfloor (y'-1)/100 \rfloor + \lfloor (y'+299)/400 \rfloor + 1) \bmod 7.$$

If the month to be displayed is m, you need to figure out whether the current year is a leap year, based on the same rules as above, and then add the number of days in months $1, \ldots, m-1$ to the day of January 1. This gives the day of the first day of month m of year y. Let us implement this:

Replace the two `printf` lines in the `main` function of `mycal.c` with

```
month = month < 0 ? 1 : month;
year = year < 0 ? 2019 : year;
first_day = find_first_day(month, year);
printf("First day = %d\n", firstday);
```

(You'll also have to declare the variable `first_day` at the beginning of your `main` function.) This ensures that not providing any month on the command line translates into the first month of the given year and not providing any year on the command line translates into 2019 for now. (You will fix this later). Then you calculate the first day of the month using the function `find_first_day`, which you will implement next, and you print its result to `stdout`.

Here's the implementation of the `find_first_day` function, which you should add anywhere before your `main` function:

```
static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int leap_year_correction(int year) {
  if (year % 400 == 0) {
    return 1;
  } else if (year % 100 == 0) {
    return 0;
  } else if (year % 4 == 0) {
    return 1;
  } else {
    return 0;
  }
}

int find_first_day(int month, int year) {
  int diff, day;

  diff = year - 1900;
  day = 1 + diff + (diff - 1) / 4 - (diff - 1) / 100 + (diff + 299) / 400;
  for (int i = 0; i < month - 1; ++i) {
    day += days_per_month[i];
  }
  if (month > 2) {
    day += leap_year_correction();
  }
  return (day % 7);
}
```

Compile the program and run it with various command line arguments and compare its output with the output of `cal` to verify that you calculate the weekday of the first day of the month correctly.

Step 9: Format a single month using `printf`

Next, you work on creating a single month-view. It should match `cal`'s output:

```
     March 2019
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
31
```

Implement this in a function `print_month`:

```
void print_month(int month, int year, int first_day) {
  ...
}
```

We build up this function slowly. Creating the day header line is easy:

```
void print_month(int month, int year, int first_day) {
  ...
  printf("Su Mo Tu We Th Fr Sa\n");
  ...
}
```

You need to precede this with a line containing the month name and the year:

```
static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

void print_month(int month, int year, int first_day) {
  ...
  printf("%s %d\n", month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  ...
}
```

To center the month and year above the list of weekdays, observe that the list of weekdays is 20 characters wide. On the header line, 5 of these 12 characters are taken up by the year and the space preceding it. Another `strlen(month_names[month - 1])` characters are taken up by the month name. Half of the remaining characters should be spaces preceding the month name:

```
static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

void print_month(int month, int year, int first_day) {
  int width;
  ...
  width = 8 + strlen(month_names[month - 1]) / 2;
  printf("%*s %d\n", width, month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  ...
}
```

Next you need to figure out how many days there are in the current month:

```
static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

void print_month(int month, int year, int first_day) {
  int width;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }
  width = 8 + strlen(month_names[month - 1]) / 2;
  printf("%*s %d\n", width, month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  ...
}
```

It remains to print the calendar view. Every day in the calendar view takes up 3 characters, two for the number of the day and one for the space between this day and the next. Thus, if the first day of the month is `first_day`, you need to start the first line of the calender view using `3 * first_day` spaces:

```
static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

void print_month(int month, int year, int first_day) {
  int width;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }
  width = 8 + strlen(month_names[month - 1]) / 2;
  printf("%*s %d\n", width, month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  printf("%*s", 3 * first_day, "");
  ...
}
```

Finally, you are ready to print the days of the current month. Every day that is not a Saturday is succeeded by a space. Every Saturday is succeeded by a newline to start the next line. This gives the following final code of the `print_month` function:

```
static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

void print_month(int month, int year, int first_day) {
  int width;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }
  width = 8 + strlen(month_names[month - 1]) / 2;
  printf("%*s %d\n", width, month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  printf("%*s", 3 * first_day, "");
  for (int i = 1; i <= days; ++i) {
    first_day = (first_day + 1) % 7;
    printf("%2d%c", i, (first_day == 0 ? '\n' : ' '));
  }
  if (first_day != 0) {
    printf("\n");
  }
}
```

Now make sure that the `main` function calls `print_month` unless the user asks for a year view. The complete code looks as follows at this point:

mycal.c (1/3)

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

void usage(char *progname) {
  printf("USAGE: %s [month] [year]\n", progname);
  exit(1);
}

static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int leap_year_correction(int year) {
  if (year % 400 == 0) {
    return 1;
  } else if (year % 100 == 0) {
    return 0;
  } else if (year % 4 == 0) {
    return 1;
  } else {
    return 0;
  }
}

int find_first_day(int month, int year) {
  int diff, day;

  diff = year - 1900;
  day = 1 + diff + (diff - 1) / 4 - (diff - 1) / 100 + (diff + 299) / 400;
  for (int i = 0; i < month - 1; ++i) {
    day += days_per_month[i];
  }
  if (month > 2) {
    day += leap_year_correction();
  }
  return (day % 7);
}
```

mycal.c (2/3)

```
void print_month(int month, int year, int first_day) {
  int width;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }
  width = 8 + strlen(month_names[month - 1]) / 2;
  printf("%*s %d\n", width, month_names[month - 1], year);
  printf("Su Mo Tu We Th Fr Sa\n");
  printf("%*s", 3 * first_day, "");
  for (int i = 1; i <= days; ++i) {
    first_day = (first_day + 1) % 7;
    printf("%2d%c", i, (first_day == 0 ? '\n' : ' '));
  }
  if (first_day != 0) {
    printf("\n");
  }
}
```

mycal.c (3/3)

```
int main(int argc, char **argv) {
  int month, year, nchars, first_day;

  switch (argc) {
    case 1:  month = year = -1;
             break;
    case 2:  month = -1;
             if (sscanf(argv[1], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    case 3:  if (sscanf(argv[1], "%d%n", &month, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || month < 1 || month > 12) {
               usage(argv[0]);
             }
             if (sscanf(argv[2], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[2]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    default: usage(argv[0]);
  }
  month = month < 0 ? 1 : month;
  year = year < 0 ? 2019 : year;
  first_day = find_first_day(month, year);
  print_month(month, year, first_day);
  return 0;
}
```

Try this code out and compare it to `cal`'s output to ensure that, modulo minor fomatting differences, they are the same.

Step 10: Query the current month and year

Before finishing off your project with a full-year view, let us deal with setting `month` and `year` to the current month and year if `mycal` is invoked without command line arguments.

To this end, you need the `time.h` header file, which provides two functions, `time` and `localtime`, (among others). The former returns the current time in seconds since "the epoch" (a standard reference date used on all Unix systems). The latter converts such a time in seconds into a structure that stores the year, month, day, hours, minutes, and seconds (and some other pieces of information).

The details of these functions can be found in their manpages. The resulting implementation of the `main function` now looks like this. The other functions remain unchanged:

```
int main(int argc, char **argv) {
  int month, year, nchars, first_day;
  time_t secs;
  struct tm *t;

  switch (argc) {
    case 1:  time(&secs);
             t = localtime(&secs);
             month = t->tm_mon + 1;
             year = t->tm_year + 1900;
             break;
    case 2:  month = -1;
             if (sscanf(argv[1], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    case 3:  if (sscanf(argv[1], "%d%n", &month, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || month < 1 || month > 12) {
               usage(argv[0]);
             }
             if (sscanf(argv[2], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[2]) || year < 1900 || year > 9999) {
               usage(argv[0]);
             }
             break;
    default: usage(argv[0]);
  }
  month = month < 0 ? 1 : month;
  first_day = find_first_day(month, year);
  print_month(month, year, first_day);
  return 0;
}
```

Try out this code to check that the output of `./mycal` is now the same as the output of `cal` if you call each without arguments.

Step 11: Add a full-year view

To print a full-year view, you need to deal with the fact that 3 month grids need to be printed side by side. There are multiple ways to address this. Here, you use a very simple strategy: you allocate a character grid, that is, a collection of strings, one per output line. You populate each line with the characters it is supposed to contain. Once this is done, you print all the lines.

Since each month grid can occupy up to 6 rows corresponding to weeks of the month plus 2 rows for the header, you need 32 rows to represent the whole year grid. Add an extra 5 rows, for a total of 37 rows, to allow one empty row above each month grid and a header row containing the year, analogous to `cal`'s output.

Each month grid is 20 characters wide. You place 3 such grids side by side and would like to allow two space characters between neighbouring grids. This increases the number of characters per row to 64. Finally, you add a newline character and an end-of-string character (`NUL`) to the end of each row. Therefore, you need 66 characters per row. The grid is thus an array `grid[37][66]`.

Next you need to modify your `main` function so it prints a year view or a month view depending on the command line arguments. In the case of a year view, you need to call the `print_month` function once for each month and you need to produce the year header at the very top of the output. You will modify the `print_month` function so that it takes a fourth parameter that indicates whether it should operate in year view (1) or month view (0) mode. This influences where on the character grid the output of `print_month` is placed. You will also change `print_month` so it returns the first weekday of the following month, which you will use as the first weekday of the current month in the next iteration. Finally, you need a function `print_grid` that outputs the constructed character grid and, once again, takes an argument that indicates whether it should produce a year view or a month view. We will discuss this function shortly. For now, the following is the final `main` function:

```
int main(int argc, char **argv) {
  int month, year, nchars, first_day;
  time_t secs;
  struct tm *t;

  switch (argc) {
    case 1:  time(&secs);
             t = localtime(&secs);
             month = t->tm_mon + 1;
             year = t->tm_year + 1900;
             break;
    case 2:  month = -1;
             if (sscanf(argv[1], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || year < 1900) {
               usage(argv[0]);
             }
             break;
    case 3:  if (sscanf(argv[1], "%d%n", &month, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || month < 1 || month > 12) {
               usage(argv[0]);
             }
             if (sscanf(argv[2], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[2]) || year < 1900) {
               usage(argv[0]);
             }
             break;
    default: usage(argv[0]);
  }

  first_day = find_first_day(month < 0 ? 1 : month, year);
  if (month < 0) {
    sprintf(grid[0] + 30, "%d", year);
    for (int i = 1; i <= 12; ++i) {
      first_day = print_month(i, year, first_day, 1);
    }
  } else {
    print_month(month, year, first_day, 0);
  }
  print_grid(month < 0);
  return 0;
}
```

The line `sprintf(grid[0] + 30, "%d", year)` refers to the character grid, whose declaration

```
char grid[37][66] = {};
```

should be placed somewhere before the `print_month` function.

The `print_month` function calculates the row and the column where the current month grid is placed based on the month and on whether a year view should be produced. In month view, the grid starts in row 0 and column 0. In year view, months 1–3 start in row 2 (following the year header and an empty line), months 4–6 start in row 11 (8 lines for the first three months plus an empty line), and so on. The months in each group of 3 start in columns 0, 22, and 44. Finally, as mentioned above, `print_month` should return the first weekday of the following month. Thus, its return type needs to be changed to `int` and a `return first_day` statement needs to be added at the end. The final `print_month` function now looks like this:

```
int print_month(int month, int year, int first_day, int year_view) {
  int width, row, col;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }

  --month;

  if (year_view) {
    row = (month / 3) * 9 + 2;
    col = (month % 3) * 22;
  } else {
    row = col = 0;
  }

  if (year_view) {
    width = 10 + strlen(month_names[month]) / 2;
    sprintf(grid[row] + col, "%*s", width, month_names[month]);
  } else {
    width = 8 + strlen(month_names[month]) / 2;
    sprintf(grid[row] + col, "%*s %d", width, month_names[month], year);
  }
  sprintf(grid[row + 1] + col, "Su Mo Tu We Th Fr Sa");
  sprintf(grid[row + 2] + col, "%*s", 3 * first_day, "");
  row += 2;
  for (int i = 1; i <= days; ++i) {
    sprintf(grid[row] + col + 3 * first_day, "%2d", i);
    if ((first_day = (first_day + 1) % 7) == 0) {
      row += 1;
    }
  }
  return first_day;
}
```

This now uses `sprintf` to place output into the character grid instead of `stdout`. Just as `sscanf` reads from its first argument, a string, instead of `stdin`, `sprintf` prints to its first argument instead of `stdout`.

Finally, you need to add the `print_grid` function. This function essentially takes the rows of the grid and prints them to `stdout`. It has to deal with two complications:

- The grid is initialized to all 0s and, so far, you have only placed characters for the day numbers in each month grid and the headers on the grid. Places that should be filled with spaces (e.g., between consecutive month grids on the same row) are still 0, which causes problems because `printf` reads this as "end of string". You need to correct this by scanning through the entire grid and changing all `NUL`-characters in columns 0–63 to spaces. Column 64 becomes a newline character. Column 65 remains `NUL` to indicate the end of the string.
- The previous description is correct if you want to print a year view. A month view is only 8 rows long and 20 columns wide. Thus, when producing a month view, you only need to substitute spaces in columns 0–19. Column 20 becomes a newline character, and column 21 becomes `NUL`. Also, in month view, you print only the first 8 rows of the grid.

This gives the following function:

```
void print_grid(int year_view) {
  int rows = year_view ? 37 : 8;
  int cols = year_view ? 64 : 20;
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < cols; ++col) {
      if (grid[row][col] == 0) {
        grid[row][col] = ' ';
      }
    }
    grid[row][cols] = '\n';
    printf(grid[row]);
  }
}
```

The following is a complete listing of the final `mycal.c` file:

mycal.c (1/4)

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

void usage(char *progname) {
  printf("USAGE: %s [month] [year]\n", progname);
  exit(1);
}

static char *month_names[] = {
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
};

static int days_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int leap_year_correction(int year) {
  if (year % 400 == 0) {
    return 1;
  } else if (year % 100 == 0) {
    return 0;
  } else if (year % 4 == 0) {
    return 1;
  } else {
    return 0;
  }
}

int find_first_day(int month, int year) {
  int diff, day;
  diff = year - 1900;
  day = 1 + diff + (diff - 1) / 4 - (diff - 1) / 100 + (diff + 299) / 400;
  for (int i = 0; i < month - 1; ++i) {
    day += days_per_month[i];
  }
  if (month > 2) {
    day += leap_year_correction(year);
  }
  return (day % 7);
}
```

mycal.c (2/4)

```
char grid[37][66] = {};

int print_month(int month, int year, int first_day, int year_view) {
  int width, row, col;
  int days = days_per_month[month - 1];

  if (month == 2) {
    days += leap_year_correction(year);
  }

  --month;

  if (year_view) {
    row = (month / 3) * 9 + 2;
    col = (month % 3) * 22;
  } else {
    row = col = 0;
  }

  if (year_view) {
    width = 10 + strlen(month_names[month]) / 2;
    sprintf(grid[row] + col, "%*s", width, month_names[month]);
  } else {
    width = 8 + strlen(month_names[month]) / 2;
    sprintf(grid[row] + col, "%*s %d", width, month_names[month], year);
  }
  sprintf(grid[row + 1] + col, "Su Mo Tu We Th Fr Sa");
  sprintf(grid[row + 2] + col, "%*s", 3 * first_day, "");
  row += 2;
  for (int i = 1; i <= days; ++i) {
    sprintf(grid[row] + col + 3 * first_day, "%2d", i);
    if ((first_day = (first_day + 1) % 7) == 0) {
      row += 1;
    }
  }
  return first_day;
}
```

mycal.c (3/4)

```
void print_grid(int year_view) {
  int rows = year_view ? 37 : 8;
  int cols = year_view ? 64 : 20;
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < cols; ++col) {
      if (grid[row][col] == 0) {
        grid[row][col] = ' ';
      }
    }
    grid[row][cols] = '\n';
    printf(grid[row]);
  }
}
```

mycal.c (4/4)

```
int main(int argc, char **argv) {
  int month, year, nchars, first_day;
  time_t secs;
  struct tm *t;

  switch (argc) {
    case 1:  time(&secs);
             t = localtime(&secs);
             month = t->tm_mon + 1;
             year = t->tm_year + 1900;
             break;
    case 2:  month = -1;
             if (sscanf(argv[1], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || year < 1900) {
               usage(argv[0]);
             }
             break;
    case 3:  if (sscanf(argv[1], "%d%n", &month, &nchars) != 1 ||
                 nchars != strlen(argv[1]) || month < 1 || month > 12) {
               usage(argv[0]);
             }
             if (sscanf(argv[2], "%d%n", &year, &nchars) != 1 ||
                 nchars != strlen(argv[2]) || year < 1900) {
               usage(argv[0]);
             }
             break;
    default: usage(argv[0]);
  }

  first_day = find_first_day(month < 0 ? 1 : month, year);
  if (month < 0) {
    sprintf(grid[0] + 30, "%d", year);
    for (int i = 1; i <= 12; ++i) {
      first_day = print_month(i, year, first_day, 1);
    }
  } else {
    print_month(month, year, first_day, 0);
  }
  print_grid(month < 0);
  return 0;
}
```

Play with it and verify that it produces the same output as the `cal` tool.

Step 12: Commit your work

Commit your work to SVN. The files you should commit are

```
lab7
├─ cmdline.c
├─ int_rep.c
├─ mycal.c
└─ type_sizes.c
```