Faculty of Computer Science, Dalhousie University *22-Oct-2018*

CSCI 2132 — Software Development

Lecture 20: Program Organization

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Generating permutations (finished)
- Multidimensional arrays as arguments
- **Program Organization:**
- Local variables

Local Static Variables

- Using keyword `static` with local variables
- Have static storage duration, but local scope
- Example:
```
int counter() {
     static int cnt = 0;
     return cnt++;
}
```
- What does the function return if we call it a few times?

18.2 External Variables (Global Variables)

We learned that local variables must be declared inside functions, which implies that there are also variables that are not declared inside any function. Indeed, in C, we can declare variables outside any functions, and these variables are called *external variables* or *global variables*. Note that variables declared inside the main function are still local variables even though the main function is special in C.

External variables have static storage duration. This means that they have permanent memory storage locations when the program is executing. Thus they retain their values throughout program execution. Previously we learned that the memory of a process is composed of four parts. Among these four parts, the *data* part is used to store external variables.

External variables have file scope. This means that they are visible from the point of declaration to the end of the enclosing file. In other words, after we declare an external variable somewhere in the program, any function defined afterwards can access this variable. Actually, we will see later that the external (i.e., global) variables have global scope; i.e., they are accessible from other files as well. If a global variable is defined with the keyword `static` then it is visible only within the file scope. It is still stored in the static memory; i.e., *data* part of the memory. Actually, we do not call it global or external variable in this case, but just static variable.

18.3 Organizing a C Program in a Single File

C is a very flexible language. It has only a limited number of rules regarding program layout: First, we cannot use a variable, type, or macro definition before we declare it or define it. Second, in C99, we cannot call a function before it has been declared or defined. Any program that follows these rules is not expected to have issues with compilers regarding its layout. This also implies that to construct a compiler for C, one has to consider many cases. This is very different from some other programming languages, such as Pascal or Ada.

This, however, does not mean that we need not pay attention to program layout at all. To develop good programming habit, you are recommended to follow the layout given below when you organize a C program in a single source file:

```
#include directives
#define directives
type definitions
global variable definitions
function prototypes (except main)
main
other function definitions
```

This layout is logical. For example, since we may use types we define to declare variables, it makes sense to write them down before we declare external variables, or define functions which may use local variables.

Example: Decimal to Binary

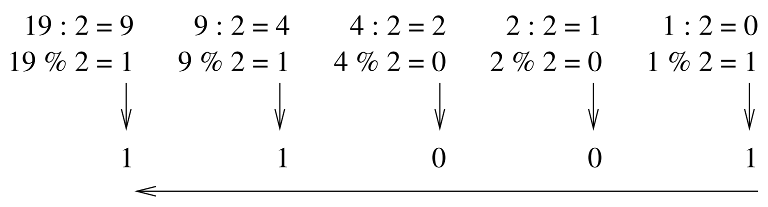
Let us organize a C program using functions as suggested above. In this example, we write a program that converts a decimal integer into a binary number. First, let us see the pseudocode for a solution that is not quite right:

```
While number > 0
   Find out the remainder after dividing the number by 2
   Print the remainder
   Divide the number by 2
```

What is wrong with the pseudocode? Let us run it using an example. Say, we want to convert 6 into a binary number. Using the above solution, we will print 011. However, the binary representation of 6 is 110. From this, we can see that our pseudocode actually prints the digits of the binary expansion in reverse order.

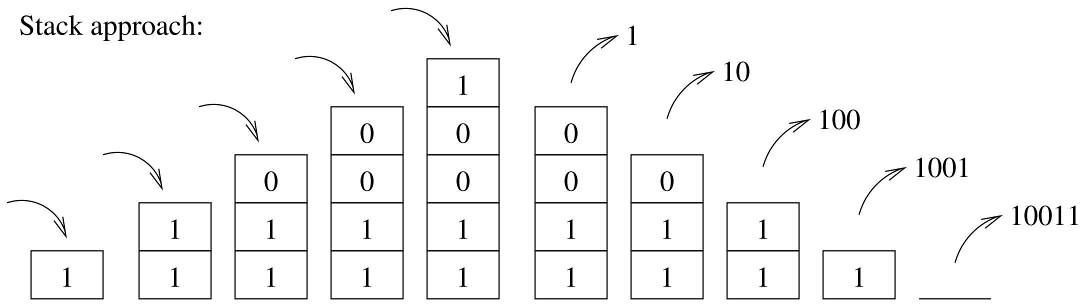
How do we fix this? We can make use of a basic data structure that we learned before. In particular, we can use a stack, since it is a last in, first out data structure. In the 3rd line of the pseudocode, instead of printing the remainder, we push it into a stack. After the loop, we pop every digit out of the stack, and print them one by one. This way we will have a correct solution.

Let us consider an example:



19 (decimal) = 10011 (binary)

Stack approach:



An implementation can be found at `~prof2132/public/decimal2binary.c-blanks`, and we will fill out the blanks.

The Fill-in-blanks Code

```
/* Program: decimal2binary.c */
#include <stdio.h>
#include <stdbool.h> /* C99 Standard */
#include <stdlib.h>

#define STACK_SIZE 100

typedef int Bit;

Bit contents[STACK_SIZE];
int top = 0;         /* index to next available spot */

void make_empty();
bool is_empty();
bool is_full();
void push(Bit i);
Bit pop();
void stack_overflow();
void stack_underflow();

int main() {
  int decimal;
  Bit bit;

  printf("Enter a decimal integer: ");
  scanf("%d", &decimal);

  while (decimal > 0) {
    bit = decimal % 2;
    __________;
    decimal /= 2;
  }

  printf("This number can be expressed in binary as: ");

  while (!is_empty())
    printf("%d", _______);

  printf("\n");

  return 0;
}

void make_empty() {
  top = 0;
}

bool is_empty() {
  return top == 0;
}

bool is_full() {
  return top == STACK_SIZE;
}

void push(Bit i) {
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

Bit pop(void) {
  if (is_empty())
    stack_underflow();
  else
    return contents[_________];
}

void stack_overflow(void) {
  printf("Error: stack overflow!\n");
  exit(EXIT_FAILURE);
}

void stack_underflow(void) {
  printf("Error: stack underflow!\n");
  exit(EXIT_FAILURE);
}
```

Pay attention to the layout of this example, and how we use external variables to implement a stack using C. Also note the exit function, which is from the header file `stdlib.h`. This function can be called to terminate a program, and its parameter is the exit code. We often use one of these two macros as arguments when calling exit: `EXIT_SUCCESS` and `EXIT_FAILURE`. The values of these macros are implementation-defined, and in most implementations, they are defined as 0 and 1, respectively, to be consistent with the convention of exit codes of most Unix utilities. Calling exit in the main function has the same effect as using the return statement. However, if we want to terminate the entire program when we execute a function that is not main, we have to call exit.

Avoid Using Global Variables Unnecessarily

Even though we used global (external) variables in the above example, we should always avoid using global variables unnecessarily. They make it harder to maintain a program: If we make changes to global variables, we have to check all the functions that are defined after we declare these variables. For similar reasons, they make it harder to debug a program.

In addition, using global variables makes it harder to reuse code. The above example works, but what if we need write a program that uses two stacks? If we use two sets of global variables, then we even need duplicate functions to get two sets of functions, one for each stack. This is certainly not good. To avoid this, we could define variables that store the data in the stack as local variables in main, and for each stack function, we declare additional parameters so that we can pass these variables as arguments to them. There is one issue here: The pop function changes the stack top variable. However, in C all arguments are passed by value. Thus, to change this variable, we should return its new value so that the statement that calls this function can assign this new value to the stack top variable. However, the pop function also need return the value popped out of the stack. Since we cannot return two values from one function, this will not work. There are ways around this, in particular by using pointers and structures, as we will see soon.

18.4 Blocks and Compound Statements

When we group statements together using curly brackets, i.e., `{` *statements* `}`, we create a block, or compound statement. We have done this before. For example, when one branch of the if statement uses more than one statement, we create a compound statement consisting of all these statements.

Even in the C standards before C99, we could have variable definitions in a block, as long as they appear at the beginning of the block. These variables have a block scope. In C99 and later, the variable definitions can appear intermixed with statements, with the scope starting there and extending until the end of the narrowest surrounding scope. The following is an example of using block variables, which is valid in older C standards as well:

```
if (i < j) {
   int temp = i;
   i = j;
   j = temp;
}
```

The variables defined in a block also have automatic storage duration. The allocation is done when the program execution enters the block, and they are deallocated when exiting the block. They have block scope, which means they are visible from the point of declaration till the end of the block.

Scope

Now we know that we can declare a variable in different places of a program. What will happen if we define variables with the same name but in different locations of a program? For this we need learn the scope rule of C.

This rule says that when blocks (here we use the term block loosely— they can even be function bodies or files) are nested, definitions in inner blocks hide those in outer blocks. Let us learn this using an example.

```
1.  int i;
2.  void f(int i) {
3.    i = 1;
4.
5.    if (i < 0) {
6.      int i;
7.      i = 4;
8.    }
9.    i = 14;
10. }
11.
12. void h() {
13.   i = 5;
14. }
```

Look at line 3. Before that, we declared i twice, one as an external variable, the other as a parameter of f. Which i are we using at line 3? Since the parameter is in the inner block, we conclude that the i in line 3 is the i defined in line 2. Following similar reasoning, we conclude that the variable i used in lines 7, 9 and 13 are the variables declared in lines 6, 2 and 1, respectively.