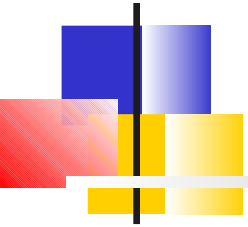


Designing Large Programs



References in our
textbook:

Section 15.1, page 349

Project 10.6, page 239

**Not all material
covered here is in the
textbook.**

To be able to

- Design a program as a collection of interacting components.
- Create a C program consisting of multiple source files.

- *Program design* means dividing the program into smaller pieces that can interact to fulfill the program's requirements.
- We have to understand the requirements before we can design the program.

- There are two kinds of parts that we must design (identify or invent.)
 - Data Structures
 - Algorithms
- An experienced designer has a large repertoire of known data structures and algorithms to draw on.
 - Normally can design a new program using known components.

- A beginner has a very limited repertoire.
 - Usually must find or invent components.
- Learning program design consists largely of building a repertoire of known components and understanding how to put them together.
- Like learning to ride a bicycle, it is best done by experience.



Implementing Large Programs

- Large programs must be divided into multiple source files.
 - Files more than a few pages long are difficult to edit, read, and understand.
 - Very large files can take a long time to compile and link.
- Let each file contain closely related code.
 - Should make sense by itself.
 - Limited number of inputs and outputs.
 - Well defined functionality.

- An Integrated Development Environment such as Visual Studio supports the development of multifile programs.
 - “Project” contains files that make up a program.
 - The IDE keeps track of what has changed and what needs to be recompiled.
 - The IDE compiles and links the program transparently.



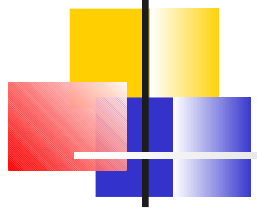
Compiling Multifile Programs

- Without an IDE, as we work on Unix, you have to say what files to compile and link.
- Unix *make* command helps.
 - “makefile” provides rules for compiling and linking.
 - You have to create the makefile with the right rules and commands for each program.
- For programs of moderate size we can simply compile everything at the same time.
 - `gcc -Wall *.c`



Designing a Large Program

- It's not practical to create an actual large program as a classroom example.
- We will simulate development of a large program with a relatively small program that still needs to be divided into parts.
 - Reverse Polish Notation Calculator
- We will add functionality to this program over several classes.



- Reverse Polish Notation (RPN) is a way to write complex mathematical expressions without using parentheses.
- Invented by the Polish mathematician **Jan Lukasiewicz** in the 1920's.
 - (wuka-SHEV-itch)



Reverse Polish Notation Calculator

- For background info see

http://en.wikipedia.org/wiki/Reverse_Polish_notation

- Or google “Reverse Polish Notation”



Reverse Polish Notation

- Operands are written first, followed by operator.
- Normal (infix) notation: $1 + 1$
- Reverse Polish notation: $1\ 1\ +$
- Infix: $1 + (2 * 3)$
- RPN: $1\ 2\ 3\ *\ +$
- Infix: $(1 + 2) * 3$
- RPN: $1\ 2\ +\ 3\ *$

- The expression is evaluated from left to right.
 - Operands and operator are replaced by result.
 - Further operations can be applied to results.

Postfix Evaluation Example

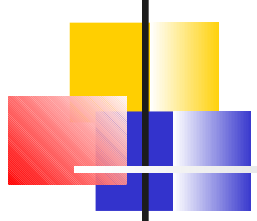
2 3 4 + 8 * 7 - +

2 7 8 * 7 - +

2 56 7 - +

2 49 +

51



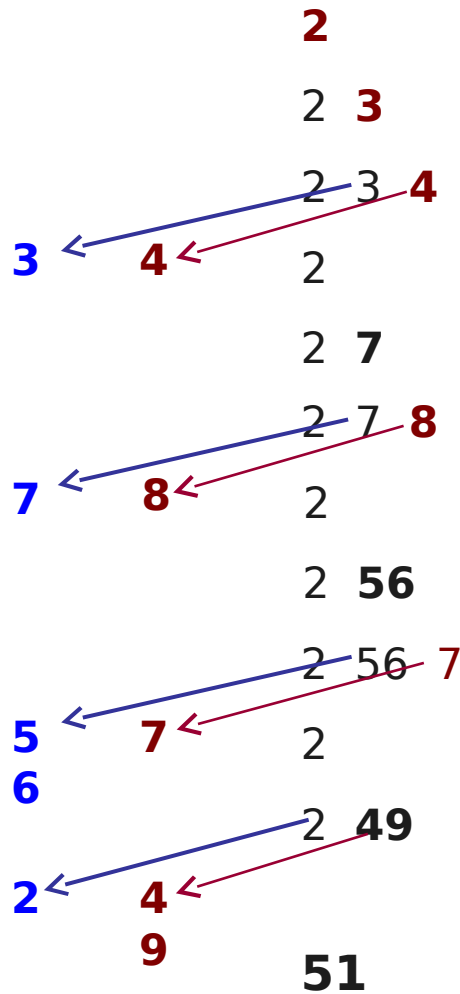
Using a Stack to Evaluate Postfix Expressions

- The stack data structure is ideal for postfix evaluation
- We will assume that all arguments are positive integers and that we will not be using the unary minus ("negative of") operator in our expressions
- The algorithm processes the expression from left to right as follows:
 - if the leftmost term is an argument (int) remove it from the expression and push it on an integer stack;
 - otherwise it is an operator symbol, so remove it and :
 - pop the top of the stack and hold that value as the right argument to the expression; then
 - pop the top of the stack and hold that value as the left argument to the expression
 - apply the appropriate operation to the arguments and push the result onto the stack
 - When the entire expression has been consumed, there will be exactly one value in the stack and it will be the value of the expression

Evaluation

left right operand stack

remaining expression



2 3 4 + 8 * 7 - +
3 4 + 8 * 7 - +
4 + 8 * 7 - +
+ 8 * 7 - +
+ 8 * 7 - +
8 * 7 - +
* 7 - +
* 7 - +
7 - +
- +
- +
+
+

- RPN can be applied to *any* arithmetic expression.
 - Any expression you can write with operators, operands, and parentheses in normal (infix) notation can be written in RPN with no parentheses.
- Why bother?
 - Cleaner, more concise representation.
 - Some errors are impossible to make.
 - RPN is simpler to parse than infix notation.

- Let's write a program to evaluate simple RPN expressions.
- To keep the program as simple as possible, it will take only integer constants as operands (no variables).
 - Operators: $+$ $-$ $*$ $/$
 - Integer arithmetic.
 - Ignore most errors.



Program Flow

- Read a string from the keyboard.
 - Should be a valid RPN expression.
- View the input string as a sequence of "tokens" separated by whitespace characters. A token is a string of non-whitespace characters.

- Example:
- if the input string is

`"25 32 15 - + 2 * = "`,

then the tokens are

`"25", "32", "15", "-", "+", "2", "*", "="`

- Utility function to extract tokens from the input string.



Program Flow

/ copies next non-whitespace string into token */*
/ return 0 if no new token available, 1 otherwise */*

```
int getNextToken(const char *buff, char *token, int len)
{
    static int j = 0; /* start of next char to be checked */
                     /* value retained from call to call */
    int i = 0;

    token[0] = 0; /* string in token is now empty -- equals "" */

    while(j < len && isspace(buff[j])) /* skip over whitespace */
        j++;

    while(j < len && !isspace(buff[j])) /* copy nonwhitespace chars into token */
        token[i++] = buff[j++];

    token[i] = 0;

    if (token[0] == 0) /* no new token */
        return 0;      /* return value treated as boolean (true/false) */
    else
        return 1;
}
```



Program Flow

- Read a string from the keyboard.
 - Should be a valid RPN expression.
- Evaluate the expression.
 - Get the tokens one after another
 - If first character of the token is a digit, use `sscanf` to input the integer in the token and push it onto the stack
 - If the first character is an operator, pop top two values off the stack and apply the operator to them. Push the result back onto the stack.
 - If char is `=`, return the result (top of the stack).
- Output the result.

- What if input is not a valid RPN expression?
 - For this example, ignore the possibility.
 - A real program would need to provide a useful error message to the user.
- Ignore characters other than whitespace, digits and operator characters.
 - Except “q” or “Q” for “quit”

- Think about how to divide the overall problem “Evaluate an RPN Expression” into smaller parts.
 - Data Structure.
 - Algorithm
- Minimal coupling
 - Parts should be independent
- Well defined functionality
 - *Responsibilities*



Divide the Program into Parts

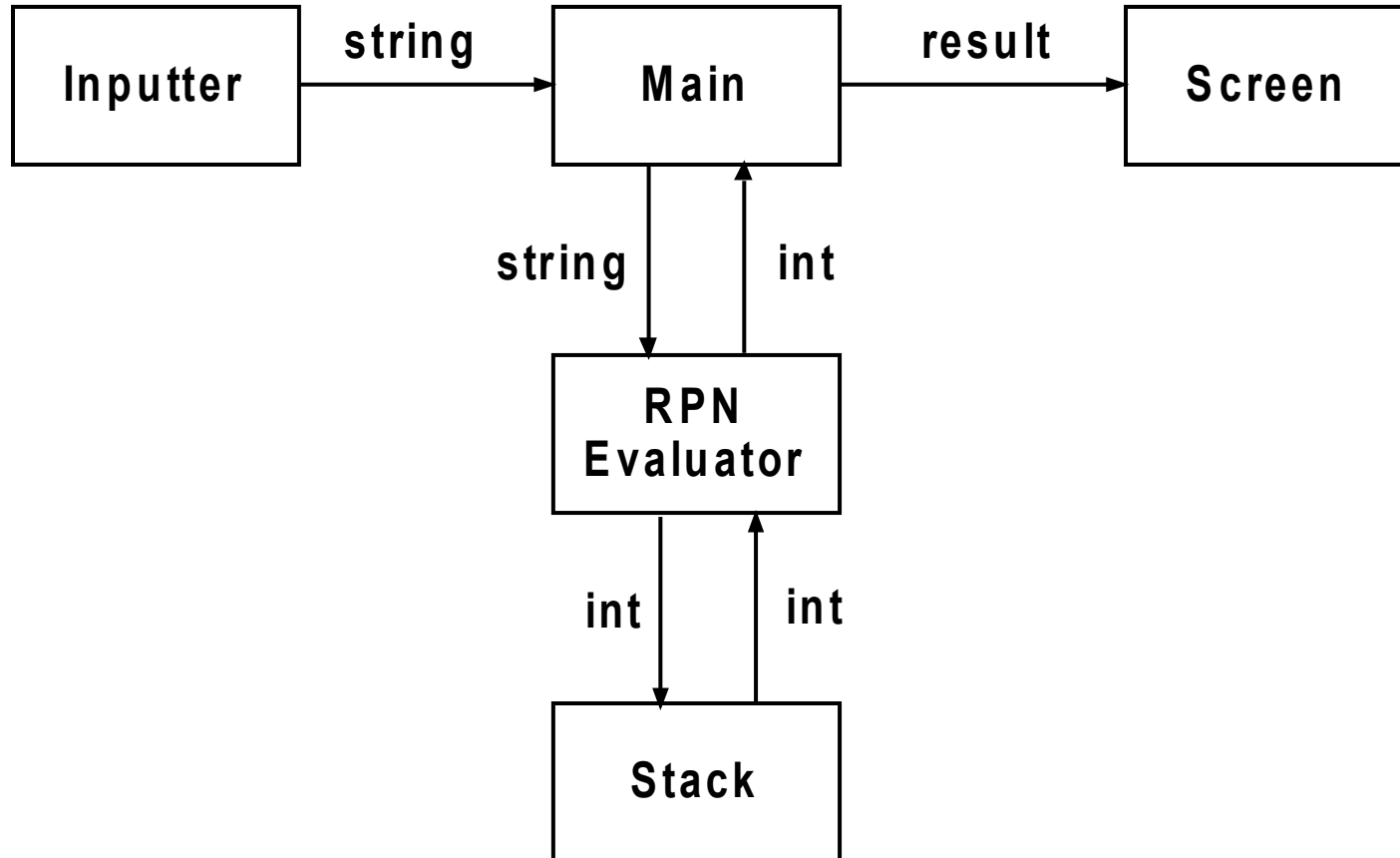
- Stack data structure.
 - Supports push and pop operations for ints.
- Inputter
 - Reads a string from the keyboard.
 - Does minimal validation.
- RPN Evaluator
 - Processes a string as an RPN expression.
 - Uses the Stack data structure.
 - Produces an integer result
- Main
 - Controls overall program execution.



Responsibilities

- Stack just provides the ability to push and pop ints.
 - Knows nothing about RPN
- Inputter just knows how to get characters from the keyboard.
 - Can validate them. Skip over invalid chars.
- RPN Evaluator is the core of the program.
 - Knows how to evaluate RPN.
- Main just coordinates the other pieces.

Design Diagram





RPN Calculator Design

- Stack data structure.
 - Supports push and pop operations for chars.
- Inputter
 - Reads a string from the keyboard.
 - Does minimal validation.
- RPN Evaluator
 - Processes a string as an RPN expression.
 - Uses the Stack data structure.
 - Produces an integer result
- Main
 - Controls overall program execution.

- Next we will implement the Inputter.
- New Items: `inputter.h`, `inputter.c`
- Only one function, *`get_input()`*
 - Prompt user.
 - Get input into buffer provided by caller.
 - Caller specifies length of buffer.
 - Provide minimal preprocessing.
 - Check for "q" (quit)
 - Keep only digits and operators and whitespace.
 - Return on "=" (Ignore anything after.)



inputter.h

```
/*  
 * This function prompts the user to enter an RPN string  
 * and reads the keyboard input into the buffer  
 * specified by the caller.  
 */  
  
void get_input(char* input_buffer, int length);
```



inputter.c

```
/* Get an RPN expression from the keyboard.  
 * Accepts q as indication that user wants to quit.  
 * Skips over invalid characters.  
 * Normal end of expression is '='.  
 */
```

```
#include <stdio.h>  
#include <ctype.h>  
#include "inputter.h"
```



inputter.c

```
void get_input(char* input_buffer, int length)
{
    int i = 0;
    char next_ch;
    printf ("Enter an RPN expression ending with =\n");
    do
    {
        next_ch = getchar();

        if (tolower(next_ch) == 'q')
        {
            // User wants to quit.
            input_buffer[0] = 'q';
            i++;
            break;
        }

        if ((next_ch == '\n') && (i > 0))
        {
            break;
        }
    }
}
```

```
    if ( isdigit(next_ch) ||
        isspace(next_ch) ||
        (next_ch == '+') ||
        (next_ch == '-') ||
        (next_ch == '*') ||
        (next_ch == '/') ||
        (next_ch == '=')      )
    {
        input_buffer[i++] = next_ch;
    }

} while ((input_buffer[i-1] != '=') && (i < length-1));

input_buffer[i] = 0;      // Provide null terminator.

while (next_ch != '\n')  // Clear keyboard input buffer
{
    next_ch = getchar();
}
}
```


- Add new item, main.c
- This file will contain our main().
 - Initially
 - Call Inputter.
 - Output string received.

```
#include <stdio.h>
#include "inputter.h"

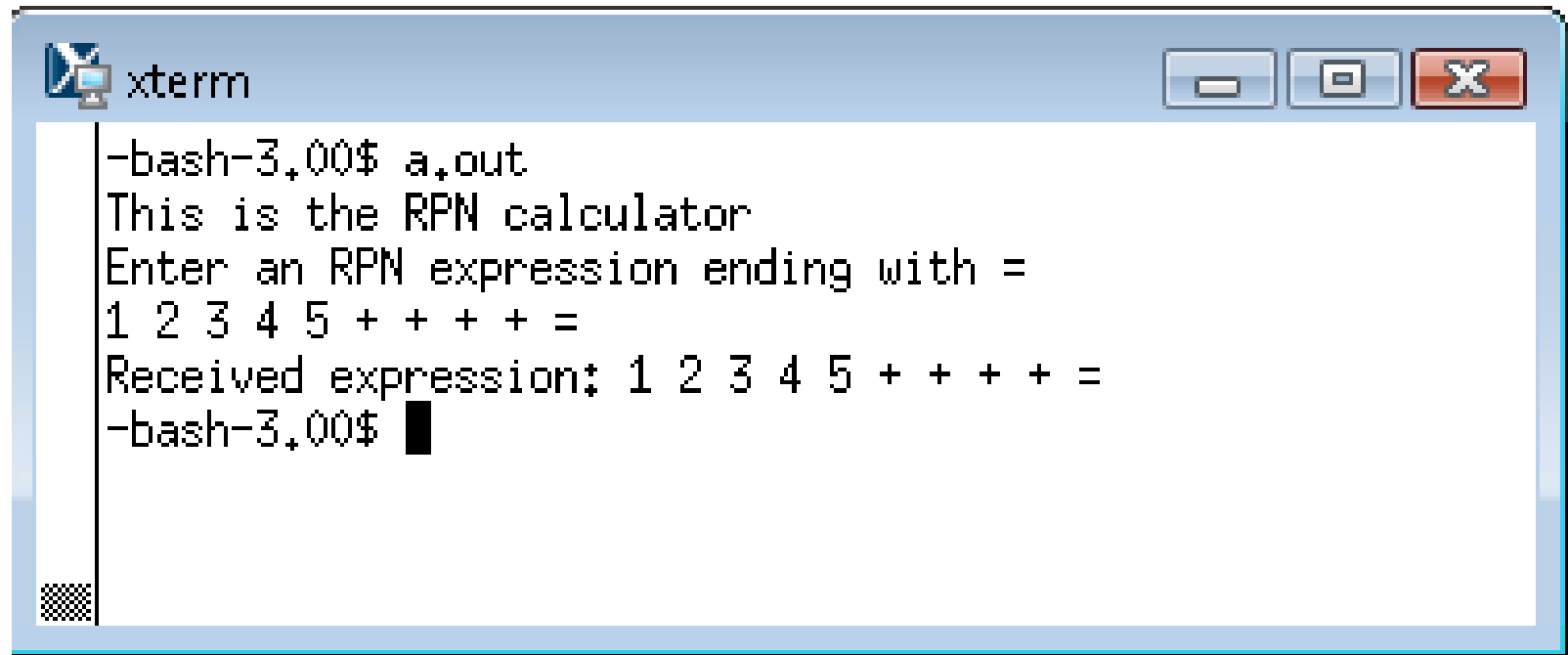
#define MAX_INPUT 1000

int main()
{
    char input[MAX_INPUT] = "";

    printf("This is the RPN calculator\n");
    get_input(input, MAX_INPUT);
    printf ("Received expression: %s\n", input);

    getchar();
    return 0;
}
```

Program Running



The image shows a terminal window titled 'xterm' with standard window controls (minimize, maximize, close). The terminal output is as follows:

```
-bash-3.00$ a.out  
This is the RPN calculator  
Enter an RPN expression ending with =  
1 2 3 4 5 + + + + =  
Received expression: 1 2 3 4 5 + + + + =  
-bash-3.00$
```

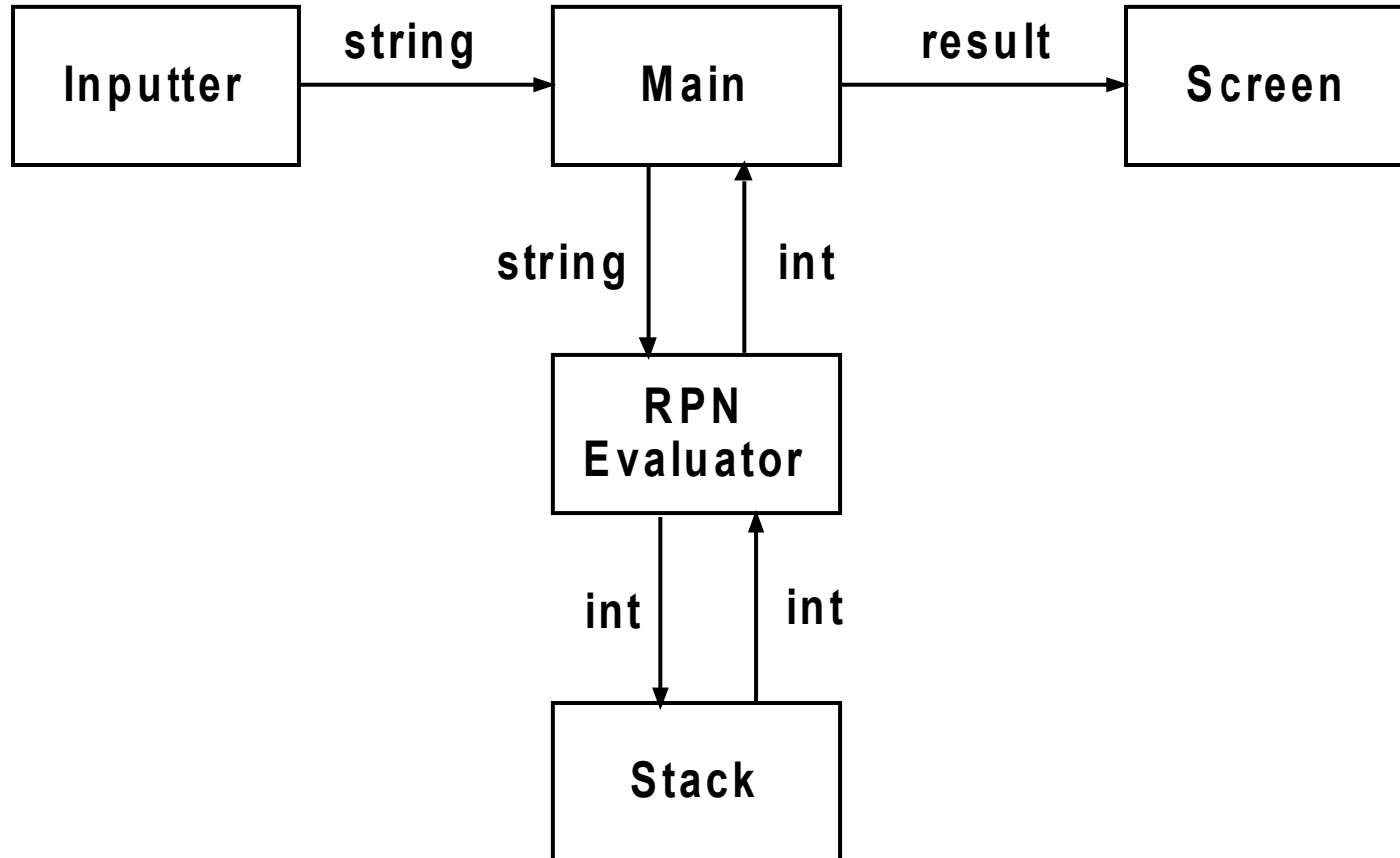
- Function calculate is responsible for evaluating RPN expressions.
- Input
 - String
 - Should consist of a single RPN expression
- Output
 - int value of the RPN expression
 - Return to caller



Limitations of Function calculate()

- We have no way for the function to tell the caller that its input is not a valid RPN expression.
- Will abort the program if the expression results in stack overflow or underflow.
 - Not good. But keep the code simple for now.
 - Push and Pop functions modified to do this
- Ignore other errors in the expression.
 - Example: Stuff left on the stack.

Design Diagram



- Function *calculate* will be divided into a number of smaller functions intended for use only by the top level function.
- To make a function invisible to the outside world, declare it *static*.
 - Same concept as for variables.
 - Limits scope to the file in which it is declared.
 - Another example of *encapsulation* or *information hiding*.

- To make a function invisible to the outside world, declare it *static*.
 - A good idea for any function that is intended only for internal use.
 - Even though its declaration is not in the header file, a function can be called from other files if not marked static.

- Will add code for calculate.c from bottom of the source file to the top.
 - Start with top level code.
 - Add function definitions above the calls.

- Add at top of calculate.c

```
#include <string.h>
#include <ctype.h>
#include "stack.h"
#include "calculate.h"
```



Top of calculate.c

```
#include <string.h>
#include <ctype.h>
#include "stack.h"
#include "calculate.h"

int calculate(const char* input)
{
    int i = 0;
    Stack S;
    int arg = 0, result= 0;
    char nextToken[50] = {}; /* holds next int string or op char */
    int length = strlen(input);

    /* continued on next slide */
```

```
while (getNextToken(input,nextToken,length))
{
    if (nextToken[0] == '=')
    {
        Pop(&result,&S);
        return result;
    }
    else if (isdigit(nextToken[0]))
    {
        sscanf(nextToken,"%d",&arg);
        Push(arg,&S);
    }
    else
    {
        do_operation(nextToken[0],&S);
    }
}

return 0;
}
```

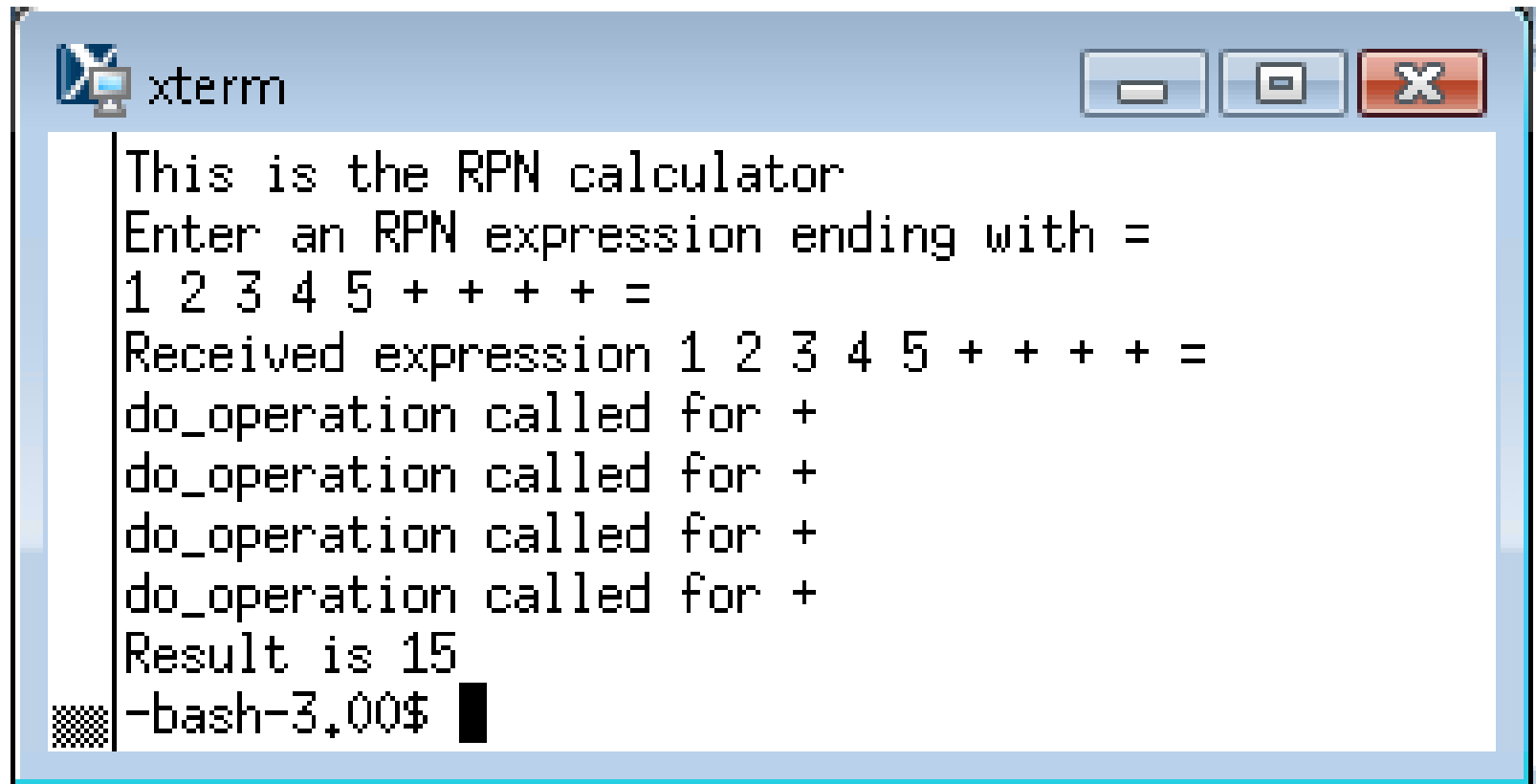


Function do_operation()

- Start with a stub:

```
/* This function performs the operation corresponding to
 * the character passed by the caller, using the top two
 * elements on the stack as operands, and pushes the result
 * onto the stack. */
static void do_operation(char ch)
{
    printf ("do_operation called for %c\n", ch);
}
```

Program Running



The image shows a terminal window titled 'xterm' with standard window controls (minimize, maximize, close). The terminal displays the following text:

```
This is the RPN calculator
Enter an RPN expression ending with =
1 2 3 4 5 + + + + =
Received expression 1 2 3 4 5 + + + + =
do_operation called for +
do_operation called for +
do_operation called for +
do_operation called for +
Result is 15
-bash-3.00$
```



Function do_operation()

```
/* This function performs the operation corresponding to
 * the character passed by the caller, using the top two
 * elements on the stack as operands, and pushes the result
 * onto the stack. */
static void do_operation(char ch, Stack *S)
{
    printf ("do_operation called for %c\n", ch);
    switch (ch)
    {
        case '+': do_plus(S); break;
        case '-': do_minus(S); break;
        case '*': do_multiply(S); break;
        case '/': do_divide(S); break;
    }
}
```



Arithmetic Operation Functions

```
static void do_plus(Stack *S)
```

```
{
```

```
    int op2, op1;
```

```
    Pop(&op2,S);
```

```
    Pop(&op1,S);
```

```
    Push (op1 + op2,S);
```

```
}
```

```
static void do_minus(Stack *S)
```

```
{
```

```
    int op2, op1;
```

```
    Pop(&op2,S);
```

```
    Pop(&op1,S);
```

```
    Push (op1 - op2,S);
```

```
}
```



Arithmetic Operation Functions

```
static void do_multiply(Stack *S)
{
    int op2, op1;
    Pop(&op2,S);
    Pop(&op1,S);
    Push (op1 * op2,S);
}
```

```
static void do_divide(Stack *S)
{
    int op2, op1;
    Pop(&op2,S);
    Pop(&op1,S);
    Push (op1 / op2,S);
}
```


- Divide large programs into small parts.
 - Small files
 - Small functions
- Each file should hold closely related functions and possibly data.
 - Should make sense by itself.
- Use conditional compilation to create different programs from the same files.
 - Example: `stack_test`
- Use *static* to make functions and variables invisible outside the file where they are defined.

- Read and understand the RPN Calculator program.
- Compile and run.