7
(Big) Program Organization

7.1 An RPN calculator

We will build a **reverse polish notation** (RPN) calculator to discuss

- Function evaluation.
- Scoping rules.
- Splitting up a program in several source files.

Recall Infix notation vs. reverse polish notation:

```
( 1 - 2 ) * ( 4 + 5 )
```

```
1 2 - 4 5 + *
```

Parentheses are not needed; the notation is unambiguous as long as we know how many operands each operator expects.

Calculator design using a stack

```
input:    1  2  -  4  5  +  *

                      5
             2     4  4  9
stack:    1  1 -1 -1 -1 -1 -9
```

Program description

- Each operand arriving is pushed on the stack
- Once an operator arrives
  - Pop apt number of operands (e.g., two for binary operators)
  - Apply operator to them
  - Push the result back onto the stack
- The value on the top of the stack is popped and printed when the end of the input line is encountered.

Calculator program algorithm

Basic algorithm of our calculator (controlling main function):

```
while (next token is not EOF)
    if (is number)
        push it
    else if (is operator)
        pop operands
        do operation
        push result
    else if (is newline)
        pop and print top of stack
    else if (is character 'q')
        end program
    else
        error
```

Program design considerations

- A function for fetching the **next input token**.
- Pushing and popping a stack are trivial, but with error handling long enough to be put each in a **separate function**.

Where to put the stack? Who should access it directly?

- Keep it in `main`.
  → Pass the stack to the routines that push and pop it.
  - But `main` doesn't need to know about the stack internals, it only uses the interface (`push` and `pop`).
- Store the stack and its pointer in **external variables**, accessible to the `push` and `pop` functions **but not** `main`.

Possible program layout in one source file

```
[declarations req'd by main]

int main(void) { /* ... */ }

[declarations req'd by push and pop: stack buffer invisible for main]

void push(double f) { /* ... */ }
double pop(void) { /* ... */ }

[declarations req'd for parsing tokens: IO functions only available from here on]

int gettoken(double *) { /* ... */ }

[declaration for IO functions with pushback buffer]

int getch(void) { /* ... */ }
void ungetch(int) { /* ... */ }
```

Marginal note

- This ordering of objects is known as **top-down design**: Start with the coarse algorithm, implement details later.
- The opposed **bottom-up design** is way more usual in C programs: Define small building blocks, and combine into `main` at the end of the source.

Source code: Calculator `main`

```c
#include <stdio.h>       /* printf(3) */
#include <stdlib.h>      /* atof(3) */

#define NUMBER '0'       /* signal that a number was found */

int gettoken(double *);  /* return value is operator, NUMBER, or EOF */
void push(double);
double pop(void);

/* reverse polish calculator */
int main(void)
{
    int type;       /* kind of input token */
    double num;

    while ((type = gettoken(&num)) != EOF) {
        switch (type) {
```

```c
        }
    }
    return 0;
}
```

```c
switch (type) {
case NUMBER:
    push(num);
    break;
case '+':
    push(pop() + pop());
    break;
case '*':
    push(pop() * pop());
    break;
case '-':
    push(-pop() + pop());
    break;
case '/':
    push(1 / pop() * pop());
    break;
case '\n':
    printf("\t%.8g\n", pop());
    break;
case 'q':
    return 0;
default:
    printf("unknown: %c\n", type);
}
```

This implementation is **erroneous**!

Can you spot the problem?

- **Order of evaluation** of function *arguments* is unknown.
  - ⇒ Which `pop()` is run first?
  - ⇒ Which stack element will be 1st/2nd argument to an operator?
- For *non-commutative* operators (`-`, `/`), we must **enforce** that the top element on the stack is used as the second argument!

```c
double num;
```

```c
case '-':
    num = pop();
    push(pop() - num);
    break;
```

- **Division by zero** is an issue, but not a major problem for `double` values.

```c
case '/':
    num = pop();
    if (num != 0.0)
        push(pop() / num);
    else
        printf("error: zero divisor\n");
    break;
```

Source code: Stack

- The **stack** itself and its fill factor (the **stack pointer**) are **shared** by push and pop
- Since they are defined outside any function, they are **external**.

```
#define MAXVAL 100

double val[MAXVAL]; /* the stack */
int sp = 0; /* next free position */

/* push x onto value stack */
void push(double x)
{
    if (sp < MAXVAL)
        val[sp++] = x;
    else
        printf("can't push %g\n", x);
}
```

```
/* pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];

    printf("stack empty\n");
    return 0.0;
}
```

- `push` and `pop` have been **declared before** `main`, but **defined after** it.
  - In between, the stack buffer was defined.
    ⇒ `main` cannot see stack internals.

- An alternative would have been:

```
/* remove top-level declarations of push and pop before main */
int main(void) {
    int type;
    double num;
    extern void push(double);
    extern double pop(void);
    ...
```

- Of course, the same holds for `gettoken`.

Source code: Read an input token

```
#include <ctype.h>  /* In general: Bad style not to put #includes at the top! */
#define MAXOP 32    /* max size of token */

int getch(void);    /* get the next character */
void ungetch(int);  /* push back one character, getch will return it next */

/* gettoken: get next operator or numeric operand */
int gettoken(double *num)
{
    int i, c;
    char buf[MAXOP + 1];           /* one for NUL */
    while (isblank(c = getch()))   /* cf. isblank(3) */
        ;
    if (!isdigit(c) && c != '.')
        return c;                  /* it's not a number, may be EOF */
```

- If the function does **not return** here, then we know **it's a number**.

⇒ Start storing the digits into the buffer.

```
    buf[0] = (char)c;
    i = 1;  /* number of digits in buffer */
    while (isdigit(c = getch())) {  /* collect integer part */
        if (i >= MAXOP) {
            printf("gettoken: number too long!\n");
            return EOF;
        }
        buf[i++] = (char)c;
    }
    if (c == '.') {
        buf[i++] = (char)c;
        while (isdigit(c = getch())) {   /* collect fraction part */
            if (i >= MAXOP) {
                printf("gettoken: number too long!\n");
                return EOF;
            }
            buf[i++] = (char)c;
        }
    }
    buf[i] = '\0';
    if (c != EOF)       /* we have to deal with that character later! */
        ungetch(c);
    *num = atof(buf);  /* store number in return parameter; cf. atof(3) */
    return NUMBER;     /* signal that we have found a number */
}
```

Can we do without `ungetch`?

It is often the case that a program cannot determine that it has read enough input until is has read too much.

Example Collecting the characters that make up a number

- Until the first non-digit is seen, the number may not be complete.
- But then the program has read one character too far.

⇒ We need to **look ahead** one character!
"Un-read" the character if we do not want to **consume** it.

Implementation We use a static extern variable to store one pushed-back character.

- `EOF` indicates that no character has been pushed back.
- `getch` reads from this variable. If `EOF`, read from *stdin*.
- `ungetch` writes to that variable[27].

[27] `ungetc`(3) declared in `<stdio.h>` un-gets a character from a given input stream

Source code: (un)getting characters

```c
int back = EOF;  /* Pushed back character, or EOF if none. */

int getch(void)  /* Get a (possibly pushed back) character. */
{
    if (back != EOF) {
        int r = back;
        back = EOF;
        return r;
    }
    return getchar();
}

void ungetch(int c)  /* Push character back on input. */
{
    if (back != EOF) {
        printf("ungetch: can only push back one char\n");
        exit(1);
    }

    back = c;
}
```

7.2 Program organisation in different files

Objective

- Divide the single source file into multiple files.
- Provide better isolation of conceptual modules.
- Allow for separate, faster compilation.

Separate compilation

- Recall that compilation is done in **phases**.
  1. Each source code file is **compiled** into object code.
  2. Object code files are **linked** into an executable

  (We have skipped some intermediate steps, *cf.* page 18, and later)

- For generating **object code**,it is not necessary that all functions and variables are **defined**.
  It is sufficient for them to be **declared** so that the compiler knows their size and lifetime!

Example Function `f` is not defined.

`main.c`

```c
#include <stdio.h>

int f(char const *);

int main(void)
{
    printf("%d\n", f("foo"));
}
```

```
$ pk-cc -c main.c
$ ls
main.c  main.o
```

With `-c` the GCC only compiles to object code!

- We are free to provide an implementation of `f` in a **separate object file**: `used.c`

```
int f(char const *c)
{
    int i = 0;
    while (*c++)
        i++;
    return i;
}
```

```
$ pk-cc -c used.c
$ ls
main.c  main.o  used.c  used.o
```

With `-c` the compiler does not require a `main` function!

- Then we **link the object files** to form an executable:

```
$ ld -o a.out -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/crt1.o /usr
/lib/crti.o main.o used.o -lc /usr/lib/crtn.o
$ ls
a.out   main.c  main.o   used.c  used.o
$ ./a.out
3
```

- The linker is fed with all the **compiled object files** for your program, including libraries and C runtime system,
- checks that all symbols, and a `main` function are defined,
- and links everything onto one executable.

- Getting the linker's arguments right depends on a lot of factors, and is hard to get right.
- Luckily, GCC does that for you:
  When gcc is called **without** -c, and sees a compiled object file, it links to the resulting binary.

```
$ pk-cc main.o used.o # only linking, no compilation
$ ls
a.out  main.c  main.o  used.c  used.o
$ ./a.out
3
```

(Actually, the gcc binary is a frontend to a bunch of relatively independent tools.)

Split the calculator into modules

- Use separate source files to better organize the code.
  - Function `main` → `calc.c`
  - The stack → `stack.c`
  - The parser → `token.c`
- Each file needs to **declare** the symbols it uses from other files.
- We also use `static` to **hide details** which are conceptually local to the module.

calc.c

```
#define NUMBER '0'
int gettoken(double *num);
void push(double x);
double pop(void);

int main(void)
{ /* definition */ }
```

stack.c

```
static double val[MAXVAL];
static int sp;

void push(double x)
{ /* definition */ }

double pop(void)
{ /* definition */ }
```

Question

- What are the benefits?
- What are the drawbacks?

token.c

```
#define NUMBER '0'
static int back = EOF;

static int getch(void)
{ /* definition */ }

static void ungetch(int c)
{ /* definition */ }

int gettoken(double *num)
{ /* definition */ }
```

This works just fine:

```
$ pk-cc -c calc.c      # produces calc.o
$ pk-cc -c token.c
$ pk-cc -c stack.c
$ pk-cc *.o    # Note: no -c flag ⇒ linking
$ ./a.out <<< '42 23/'
        1.826087
```

- ☼ Isolation of concepts ⇒ reusable code.
- ☼ If one module changes, only the depending files need **recompilation**.
- `NUMBER` is defined repeatedly.
- In fact, each file using, *e.g.*, `token.c` must **repeat** the declarations of `push` and `pop`.
  ⇒ Hard to maintain correctly!

Solution We have a **tool** do this for us:

- The C Preprocessor (*cf.* page 191) can `#include` a source file into another one.
- Put the shared declarations into a so called **header file** (suffix `.h`).
- `#include` this file in each `.c` file which **uses** these declarations.
- Also, `#include` this file in the **defining** source, to be warned about inconsistencies.

⇒ The header file serves as an **interface description**, listing the objects **provided** by a module.

Including source code

stack.h

```c
void push(double);
double pop(void);
```

stack.c

```c
#include "stack.h"
#include <stdio.h>

#define MAXVAL 100
static double val[MAXVAL];
static int sp;

void push(double x) { ... }
double pop(void) { ... }
```

calc.c

```c
#include <stdio.h>

#include "token.h"
#include "stack.h"
int main(void) { ... }
```

token.h

```c
#define NUMBER '0'
int gettoken(double *num);
```

token.c

```c
#include "token.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define MAXOP 32

static int back = EOF;

static int getch(void) { ... }
static void ungetch(int c) { ... }
int gettoken(double *num) { ... }
```

- Difference between
  - `#include <file>` — look for include file in a standard list of system directories. Can be modified with GCC's `-I` flag.
  - `#include "file"` — look for file in the directory of the including file, fall back to a user defined list, then to the list used by `#include<...>`
- The filename **must not contain** any of `>`, `\n`, `"`, `'`, `~`, `/*`.
- How to avoid loops?

```
$ cat foo.h
#include "bar.h"
$ cat bar.h
#include "foo.h"
$ cat main.c
#include "foo.h"
```

```
$ pk-cc main.c
In file included from bar.h:1:0,
                 from foo.h:1,
                 # ...many repetitions...
foo.h:1:17: error: #include nested too deeply
 #include "bar.h"
```

⇒ We need to make sure that each header file is included only once!

7.3 The C Preprocessor

- As an early **compilation phase**, the preprocessor is called automatically by the compiler.
- The preprocessor **modifies the source** code before compilation.
  - Inclusion of named files (by `#include`).
  - Macro substitution (defined with `#define`).
  - Conditional compilation (*cf.* page 202).
- Documentation is available online[28] with the other GCC manuals, or via `info cpp`, and `cpp`(1).
- We have already discussed file inclusion (*cf.* page 190).
  Avoiding cyclic definitions is explained on page 204.

[28] http://gcc.gnu.org/onlinedocs/gcc-4.8.2/cpp/

Simple macro definition

- A directive of the form

```
#define name token...
```

causes the preprocessor to replace *subsequent* occurrences of the **token** name with the given sequence of tokens.

- CPP does not replace within **string literals**, or **comments**.

Warning CPP performs simple **textual substitution** only.

```c
#include <stdio.h>

#define  x  1 + 2

int main(void)
{
    printf("%d\n", 2*x);
    return 0;
}
```

```
$ pk-cc main.c
$ ./a.out
4                          # yes: four!
```

- What has happened here?
- How can we solve this?

Solution Put the replacement text into parenthesis:

```
#include <stdio.h>

#define  x   (1 + 2)

int main(void)
{
    printf("%d\n", 2*x);
    return 0;
}
```

```
$ pk-cc main.c
$ ./a.out
6
```

- You can have a look at the preprocessor output with `gcc -E main.c`, or you can run `cpp` as a standalone program.

Macros with arguments

- A directive of the form

```
#define name( identifier[,identifier] ) token...
```

where there is **no space** between the name and the '(', is a macro definition with parameters given by the identifier list.

Example

```
#define isupper(c) ((c) >= 'A' && (c) <='Z')
```

- Why are there so many parenthesis?
- Why is there no ; at the end?

Example

Avoid the overhead of a function call ⇒ faster?

```
#define square(x) ((x) * (x))
double y = square(read_num_from(stdin));
```

- What do you think?

Stringification

- When a macro **parameter** is used with a leading #, it is replaced with the literal text of the argument, converted to a string literal.
- This only works *in the body* of a macro definition.

```c
#define SHOW(type) \
    printf("%s\t%zu\n", #type, sizeof(type))

int main(void)
{
    SHOW(int);
    SHOW(double);
    return 0;
}
```

```
$ pk-cc main.c
$ ./a.out
int     4
double  8
```

Notes

- Macro definitions may be split into lines with \newline.
- Two **consecutive string literals** will be concatenated into one:

```c
#define SHOW(type) printf(#type "\t%zu\n", sizeof(type))
```

Concatenation

- Normally, CPP operates at the **granularity** of C tokens.
  (That's why the input should be lexically valid C code)
- The ## operator allows to **concatenate** two tokens, when used in a macro body.

Example

```c
struct command {
    char *name;
    void (*function) (void);
};

struct command commands[] = {
    { "quit", quit_command },
    { "help", help_command },
    { "calc", calc_command },
    /* ... */
};
```

```c
struct command {
    char *name;
    void (*function) (void);
};

#define COMMAND(NAME) \
    { #NAME, NAME ## _command }

struct command commands[] = {
    COMMAND(quit),
    COMMAND(help),
    COMMAND(calc),
    /* ... */
};
```

Careful with compound macros!

```
#include <stdio.h>

#define SHOW(type) \
    count++; \
    printf("%d\t" #type "\t%zu\n", count, sizeof(type))

int main(void)
{
    int count = 0;

    SHOW(int);
    SHOW(double);

    if (42 < 23)
        SHOW(char);

    return 0;
}
```

Question What will happen? Why? How to solve this?

Try braces around the macro's body:

```c
#include <stdio.h>

#define SHOW(type) {        \
        count++; \
        printf("%d\t" #type "\t%zu\n", count, sizeof(type)); \
    }

int main(void)
{
    int count = 0;

    if (42 < 23)
        SHOW(char);

    if (99 < 1)
        SHOW(double);
    else
        SHOW(float);

    return 0;
}
```

Question This won't even compile! Why?

Solution Make the compound a statement: Use a `do-while` block.

```
#include <stdio.h>

#define SHOW(type) do {       \
        count++; \
        printf("%d\t" #type "\t%zu\n", count, sizeof(type)); \
    } while (0)

int main(void)
{
    int count = 0;

    SHOW(int);

    if (42 < 23)
        SHOW(char);

    if (99 < 1)
        SHOW(double);
    else
        SHOW(float);

    return 0;
}
```

Predefined macros

Several macros are **predefined**. They cannot be undefined or redefined.

`__LINE__` A decimal constant containing the current source line number.

`__FILE__` A string literal containing the name of the file being compiled.

`__DATE__` A string literal containing the date of compilation.

`__TIME__` A string literal containing the time of compilation.

`__STDC__` The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations.

Example

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

#define ASSERT(a) do { if (!(a)) { \
        fprintf(stderr, \
        __FILE__ ":%d: Assertion " #a " failed\n", __LINE__); \
        exit(1); \
    } } while (0)

int main(void)
{
        ASSERT(1 < 2);
        ASSERT(23 > 42);

        return 0;
}
```

```
$ pk-cc cpp5-assert.c
$ ./a.out
cpp5-assert.c:13: Assertion 23 > 42 failed
```