Matheus Augusto de Souza
118164136
mataugusto1999@gmail.com

(*) Uma breve apresentação da calculadora polonesa inversa.

A calculadora polonesa é uma forma de efetuar cálculos matemáticos básicos (soma, subtração, multiplicação e divisão) colocando primeiro o operador e depois os valores a serem calculados.
Por exemplo, convencionalmente escrevemos uma soma de valores como sendo ‘1 + 1’ com o resultado sendo 2, no formato da calculadora polonesa, essa soma ficaria ‘+ 1 1 ’.
Então a calculadora polonesa inversa vai ser esta soma ao contrário, ou seja, ‘1 1 +’.

(*) A pilha.

Antes de vermos um problema temos que ter um entendimento básico do conceito de pilha (stack).
A pilha é um conjunto de elementos sobrepostos, como uma pilha de livros por exemplo, e para se retirar algum elemento isso deve ser feito em ordem , ou seja, o último elemento adicionado deve ser o primeiro a ser retirado (pois está acima dos outros) e o primeiro elemento adicionado vai ser o último a ser retirado (pois está abaixo de todos os outros).
Por essa razão chamamos esse processo de pilha, por ser uma forma de empilhar e desempilhar elementos.

(*) Como podemos explicar a seguinte operação?

```
@echo '2 3 4 + *' | ./polonesa.exe
        14
```

Então, podemos observar essa pilha como:
[2][3][4] onde temos os seguintes operadores ‘ + * ’. Então, devemos fazer a operação seguindo a ordem da direita para a esquerda, já que os elementos à direita são os superiores e os à esquerda os inferiores, logo temos que a primeira operação a ser realizada deve ser ‘3 4 *’, [3][4]* cujo resultado é [12].
Antes da primeira operação, temos a retirada dos elementos 4 e 3 da pilha respectivamente, resultando apenas no elemento [2] na pilha e, depois da multiplicação desses, a adição do elemento 12 na pilha, gerando uma pilha com os elementos [2][12], então agora será resolvida a operação ‘2 12 +’ cujo resultado é [14].

Antes dessa operação ocorrer são retirados os elementos 12 e 2 da pilha e após a soma ter sido feita é adicionado o elemento resultante à pilha.

(*) Como pode código escrito em main.c conseguir acessar nomes definidos em outros arquivos através do uso do qualificador '/extern/'?

Quando uma variável global é criada, qualquer arquivo pode ter acesso à ela através do uso do qualificador extern, mas não podemos atribuir valores a essas variáveis externas, pois, caso façamos isso, estaríamos declarando dois valores a uma variável e isso resultará em um erro, assim como não podemos ter duas variáveis com o mesmo nome dentro de um projeto.
Caso não queiramos que a variável global criada possa ser utilizada por outros arquivos, devemos utilizar o qualificador static. Com esse qualificador deixamos nossa variável com acesso restrito apenas para o arquivo em que ela foi definida.

(*) O que fazem os procedimentos push() e pop() em stack.c?

```
int sp = 0;
int val[MAXVAL];

void push(int n) {
  if (sp < MAXVAL)
    val[sp++] = n;
  else
    printf("error: stack full: %d\n", n);
}
```

Nesse arquivo é definido as variáveis inteiras sp e val[ ], onde sp é o ponteiro da pilha (marca o topo da pilha, a próxima posição livre da pilha) e val é a pilha com os seus elementos, que são utilizadas pelo /extern/ em main.c.
A função push é responsável por adicionar um elemento na pilha, este elemento é inserido na parte superior da pilha, isto é, no fim dela, a função recebe um inteiro como argumento e se o ponteiro for menor que o tamanho máximo da pilha o valor inteiro de n é adicionado à pilha val[ ] na posição sp (que começa em 0) e após isso, o valor de sp é incrementado (assumindo a próxima posição livre, neste caso, a posição 1).
Por exemplo, caso o valor de sp seja 0 e n seja 9, teríamos que:
val[sp++] = n => val[0] = 9 e logo após isso o valor de sp passará a ser 1.
Quando utilizamos val[sp++] = n, primeiro é atribuído o valor de n para val na posição sp na pilha, isto é val[sp] = n, após o elemento ser atribuído, é incrementado o valor de sp.
Caso a notação fosse val[++sp] = n, o caminho seria o inverso, primeiro seria incrementado para só depois o valor ser atribuído, o que mudaria a posição do elemento n.

Quando o valor de sp se iguala ao de MAXVAL a pilha para de ser preenchida e é retornado um erro avisando que a pilha está cheia, bem como o elemento que iria ocupar o espaço em questão na pilha.

```
int pop(void) {
  if (sp > 0)
    return val[--sp];
  else {
    printf("error: tried to pop an empty stack\n");
    return 0;
  }
}
```

O procedimento pop() é responsável por tirar o último elemento adicionado, pois este é o elemento que está acima dos outros elementos na pilha. Neste procedimento, caso o valor de sp seja maior que 0 irá retornar o elemento na posição val[--sp], o que quer dizer que primeiro irá decrementar o valor do ponteiro sp e retornar o valor desta posição ([sp-1]).
Por exemplo, caso o valor de sp seja 1 o elemento retornado estará na posição 0 da pilha, ou seja, val[0].
Caso a notação fosse val[sp--] o processo seria o contrário, primeiro mostraria o elemento na posição de val[sp] para depois decrementar o valor e sp.
Caso o valor de sp seja 0, retornará um erro avisando que a pilha que tentou abrir está vazia. Isso ocorre porque não existe nenhum elemento para ser retirado da pilha.

(*) O mecanismo de /buffer/.

O procedimento getch é o responsável por remover os elementos de buf e o procedimento ungetch é o responsável por adicionar os elementos ao array buf.

```
char buf[BUFSIZE];
int p = 0;
int getch(void) {
  if (p > 0)
    return buf[--p];
  else
    return getchar();
}
```

Nesse arquivo é definido primeiro um array chamado ‘buf’ de tamanho BUFSIZE e uma variável p que define a próxima posição livre em buf com valor inicial igual a 0. Definimos também getch cujo retorno será um valor inteiro e tem entrada vazia. Caso p seja maior que 0 retornará o valor de p é decrementado e depois é retornado o elemento de buf na posição (p-1). Quando p chega ao valor de 0 é retornado getchar, o que vai ler a entrada padrão.

```
void ungetch(int c) {
  if (p >= BUFSIZE)
    printf("ungetch: too many characters; buffer full\n");
  else
    buf[p++] = c;
}
```

Agora definimos ungetch que tem como retorno vazio e como parâmetro de entrada um inteiro c. Se o valor de p for maior ou igual a BUFSIZE é avisado que o buf já está cheio, e assim não pode mais ser adicionados novos elementos ao array. Caso p seja menor que BUFSIZE é adicionado o valor de c à posição p no array buf e após isso é incrementado o valor de p, ou seja, buf[p] = c e após isso fazemos p+1.

(*) Somos obrigados a usar um mecanismo de buffer nesse programa?

Sim, o mecanismo de buffer está fazendo o mesmo processo que os mecanismos pop() e push(), mudando somente de uma pilha para um array mas o processo de buffer é essencial para o funcionamento de getop, já que este trabalha com array.

(*) O procedimento /getop/.

O procedimento getop é responsável por separar os tipos de informações, separando valores de operadores e de dígitos.

```
int getop(char s[]) {
  int i, c;

  while ((s[0] = c = getch()) == ' ' || c == '\t')
    ;

  s[1] = '\0';
```

Definimos i e c como dois valores inteiros. O laço while ignora espaços em branco e tabs e adiciona os valores diferentes de espaços e tabs ao array s. Após adicionar o valor c (diferente de de tabs e espaços em branco) é adicionado o fechamento ao array.

```
  if (!isdigit(c)) {
    return c;
  }
```

Se c não for um dígito ou um inteiro é retornado o valor de c. O objetivo é separar os operadores, que serão responsáveis pelos cálculos matemáticos, e adicionar somente os dígitos ao array.

```
  i = 0;
```

```
  if (isdigit(c))
    while (isdigit(s[++i] = c = getch()))
      ;
  s[i] = '\0';
```

Primeiro inicializamos i como sendo 0. Caso c seja um dígito, então o valor de c vai ser adicionado ao array s na posição i+1 (pois, antes de assumir a posição i, o valor de i é incrementado) e esse c passa a assumir o local onde anteriormente havia um fechamento de array. após isso é retirado o elemento da entrada. O while reconhecerá como sendo um valor inteiro quando encontrar espaços em branco, ou seja, até que ele encontre um espaço em branco, os dígitos vão sendo entendidos como um único inteiro. Por exemplo: 122 3 55 são três valores inteiros diferentes. Após o inteiro ser reconhecido é adicionado o fechamento do array.
O operador também aparece nesse array, mas é logo substituído pelo fechamento.

```
  if (c != EOF)
    ungetch(c);

  return NUMBER;
}
```

Se c for diferente do fim do arquivo, ungetch retorna o elemento c para assumir um valor dado pelo usuário. O operador também é retornado aqui, pois não é o fim do arquivo, isso é um dos motivos do arquivo main conseguir reconhecer o operador utilizado no fim.
No fim desses processos é retornado o valor NUMBER.

(*) O procedimento /main/.

O procedimento main é o arquivo principal, responsável por receber a entrada padrão, ou seja, é através do arquivo main que todos os procedimentos serão utilizados para a aplicação da calculadora polonesa inversa.

```
int main(void) {
  int type; int op2; char s[MAXOP];

  while ( (type = getop(s)) != EOF ) {

    switch (type) {
    case NUMBER:
      push(atoi(s));
      break;
```

Enquanto o getop não retornar EOF, esse while continuará funcionando. Essa é a razão de precisarmos selecionar Ctrl+D para encerrar a aplicação da calculadora polonesa quando utilizamos de forma interativa.

(*) O que faz o procedimento atoi()?

A função atoi vai converter o array s em número inteiro, o qual será utilizado na calculadora. Podemos ver isso porque o push iria adicionar o valor do array á uma pilha, o que não faria muito sentido ter uma pilha com um array, então o procedimento atoi() transforma esse array em um número inteiro que é adicionado à pilha.

(*) Por que na subtração (bem como na divisão) precisamos necessariamente fazer:

```
  op2 = pop();
```

antes de passar a subtração para push()?

```
case '-':
      op2 = pop();
      push(pop() - op2);
      break;
```

Para que não tenhamos um problema de ordem (visto que pop() - pop() é uma subtração que pode assumir duas ordens e que pode ter resultados diferentes), precisamos definir um dos pop() como alguma variável, no nosso caso, decidimos definir o valor a ser subtraído (o segundo valor) como op2, para que não tenhamos um problema, como 2 - 3 = -1 que é diferente de 3 - 2 = 1.

Para a divisão podemos tomar uma explicação semelhante.

```
case '/':
      op2 = pop();
      if (op2 != 0) {
        push(pop() / op2);
      } else {
        printf("error: division by zero is *undefined* by
mathematics\n");
      }
      break;
```

Primeiro devemos deixar restrita a divisão, já que não existe divisão por 0.
Como no caso da subtração, definimos o segundo valor (o divisor) como op2 para não ter nenhuma troca de valores que possam mudar o resultado esperado, já que 2/3 é diferente de 3/2.

(*) O que poderia acontecer se tivéssemos feito como na adição e multiplicação?

Poderíamos ter valores diferentes para uma mesma operação e poderíamos até mesmo não achar um resultado real, como é o caso de divisão por 0.