Apêndice A

Tratamento de exceções

Em C, é possível emular o tratamento de exceções, tal qual linguagens de alto nível, como C++, Java, Python, etc. Analisaremos abaixo algumas das formas de tratar exceções em C.

As formas de fazê-lo são usando recursos de quebra de fluxo do programa, provenientes da própria linguagem. No link https://www.vivaolinux.com.br/artigo/Tratamento-de-excecoes-na-linguagem-C, do autor Vinícius dos Santos Oliveira, é explicado com mais propriedade e detalhes as formas de implementar o tratamento de exceções mostrados nesta seção.

Usando *goto*

A quebra de fluxo do programa pode ser feita usando o comando *goto*. Esta forma preza pela simplicidade, mas tem a limitação de poder ser usada apenas dentro de um bloco de código.

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.
4.  int main(void){
5.     FILE* fp;
6.       if(fp)
7.          goto try;
8.       else
9.          goto catch;
10.
11.    //try
12.    try:
13.      fp = fopen(“/home/user/file.txt”, “a”);
14.      goto finally;
15.
16.    //catch
17.    catch:
18.       puts(“Erro ao abrir arquivo”);
19.      goto finally;
20.
21.    //finally
22.     fclose(fp);
23.     return EXIT_SUCCESS;
24. }
25.
```

Usando setjmp/longjmp

Diferente de *goto*, que é limitada apenas dentro de um bloco de código, as funções setjmp e longjmp permitem maior flexibilidade, uma vez que, a grosso modo, a função setjmp salva um ponto específico no código durante a execução do mesmo, retornando a este ponto posteriormente, através do uso da função longjmp. Como este ponto é salvo em tempo de execução, a flexibilidade supracitada consiste no fato de que, usando estas funções, é permitido um “salto” entre diferentes blocos de código.

```
1.  /* Este é o exemplo anterior, usando setjmp e longjmp */
2.  #include<stdio.h>
3.  #include<stdlib.h>
4.  #include<setjmp.h>
5.
6.  void openFile(jmp_buf catch){
7.     FILE* fp;
8.     fp = fopen("/homer/seth/myFile.txt", "r");
9.       if(fp == NULL){
10.   /* retorna ao ponto salvo por setjmp, “lançando a exceção” */
11.        longjmp(catch, 1);
12.     }
13.     else if(fp != NULL){
14.        puts("Arquivo aberto com exito");
15.    }
16. }
17.
18. int main(void){
19. jmp_buf try;
20.   if(setjmp(try) == 0){
21.     openFile(try);
22.   }
23.   /* lança a exceção */
24.   else{
25.     puts(“Erro ao abrir o arquivo”);
26.   }
27. return EXIT_SUCCESS;
28. }
```

No exemplo acima, faz-se mister o uso do *header* setjmp.h, para o uso das funções setjmp e longjmp. Na linha 18 é declarado o identificador try – poderia ser qualquer nome, este foi usado para fins didáticos – do tipo jmp_buf. A função setjmp é chamada na linha 19, cujo parâmetro é o identificador de nome try. Como ela sempre retorna zero, este bloco é executado, e a função openFile é chamada. (linha 20)

A função openFile recebe como argumento o identificador do tipo jmp_buf. Ao tentar abrir um arquivo, usando um ponteiro do tipo FILE através da função fopen, ocorre uma exceção (linhas

8 a 11) e a função longjmp é chamada, retornando ao ponto do código que foi “salvo” anteriormente por setjmp. A exceção – proposital, neste exemplo -, ocorre porque, por padrão, não existe diretório /homer nos sistemas *nix like :P

Ao retornar novamente no ponto “salvo” por setjmp, longjmp, ao ser chamada, passa como segundo argumento o valor 1 (linha 11). Qualquer valor diferente de zero seria válido. Como setjmp agora retorna um valor diferente de zero, o segundo bloco de código é executado, simulando o disparar de uma exceção. (linha 24 e 25)

Outro efeito interessante que pode ser obtido fazendo do uso destas funções é o fato de longjmp reestabelecer o fluxo do programa atual – i.e., retornar à parte do programa que foi salva por setjmp – independentemente dos vários níveis de chamadas de função. Observe o código abaixo:

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<setjmp.h>
4.  jmp_buf jmp;
5.
6.
7.  void printMessage(){
8.     printf("Olá ");
9.     longjmp(jmp, 1);
10. }
11.
12. void chama4(){
13.    printMessage();
14. }
15.
16. void chama3(){
17.    chama4();
18. }
19.
20. void chama2(){
21.    chama3();
22. }
23.
24. void chama1(){
25.    chama2();
26. }
27.
28. int main(void){
29.
30. if(setjmp(jmp) == 0){
31.    chama1();
32. }
33. printf("mundo\n");
34.    return EXIT_SUCCESS;
```

```
35. }
```

```
Olá mundo
```

No exemplo acima, após as sucessivas chamadas de função (linhas 30, 24, 20, 16 e 12), a função longjmp (linha 8) retorna imediatamente na parte do código que foi salvo por setjmp. (linha 29)

Observe a saída do compilador*:

* Compilador: GNU gdb (Debian 7.12-6) 7.12.0.20161007-git

```
Breakpoint 1, main () at ex5.c:30
30      if(setjmp(jmp) == 0){
(gdb) s
31              chama1();
(gdb) s
chama1 () at ex5.c:25
25      chama2();
(gdb) s
chama2 () at ex5.c:21
21      chama3();
(gdb) s
chama3 () at ex5.c:17
17      chama4();
(gdb) s
chama4 () at ex5.c:13
13      printMessage();
(gdb) s
printMessage () at ex5.c:8
8       printf("Olá ");
(gdb) s
9       longjmp(jmp, 1);
(gdb) s
Olá mundo
[Inferior 1 (process 24688) exited normally]
```

Como é sabido, quando há várias chamadas de funções, as mesmas são “empilhadas umas sobre as outras”. De acordo com a Wikipedia:

Em ciência da computação, **LIFO** (acrônimo para a expressão inglesa ***Last In, First Out*** que, em português significa **último a entrar, primeiro a sair**) refere-se a estruturas de dados do tipo pilha. É equivalente a **FILO**, que significa ***First In, Last Out*** . (https://pt.wikipedia.org/wiki/LIFO)

Mas, ao analisar o código, se pode constatar que as funções são chamadas e empilhadas na pilha (linhas 31, 25, 21, 17 e 13), mas algo interessante acontece na linha 9 do código. Após a função printMessage chamar a função printf(linha 8) e consequentemente longjmp(linha 9), ao seu

término, o comportamento esperado era o de as funções serem retiradas da pilha, uma a uma, como citado anteriormente, mas o que acontece é que o controle é devolvido imediatamente à função main, na linha 33, onde há uma chamada à função printf.

Após esta análise, o autor conclui que, com o uso das funções setjmp e longjmp, é possível "pular" entre blocos de funções, eliminando o *overhead* de chamadas entre elas. (neste caso) Outra observação importante é que se deve ficar atento ao usar esta forma, afim de evitar *memory leaks*. (vazamentos de memória)

Exceções associadas a inteiros

Dando continuidade às formas de se emular o tratamento de exceções utilizando a linguagem C, ao associarmos as exceções a números inteiros, conseguimos fazer o tratamento de tais exceções de forma mais adequada. Observe o exemplo abaixo:

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<setjmp.h>
4.
5.  int soma(int x, int y, jmp_buf throw){
6.     if(x < 0 && y < 0){
7.       longjmp(throw, 3);
8.     }
9.     if(x < 0){
10.       longjmp(throw, 1);
11.    }
12.    else if(y < 0){
13.       longjmp(throw, 2);
14.    }
15.    else{
16.       return x + y;
17.    }
18. }
19.
20. int main(void){
21. int n1, n2, result;
22. jmp_buf bff;
23. puts("Programa soma apenas números positivos");
24. puts("Insira o primeiro número: ");
25. scanf("%d", &n1);
26. puts("Insira o segundo número: ");
27. scanf("%d", &n2);
28.
29.    switch(setjmp(bff)){
30.       case 0:
31.          result = soma(n1, n2, bff);
32.          printf("\nResultado da soma: %d\n", result);
```

```
33.         break;
34.       case 1:
35.         puts("Não foi possível realizar a soma. Primeiro número deve ser positivo");
36.         break;
37.      case 2:
38.         puts("Não foi possível realizar a soma. Segundo número deve ser positivo");
39.         break;
40.       case 3:
41.         puts("Não foi possível realizar a soma. Ambos os números devem ser positivos");
42.         break;
43.       default:
44.         puts("Erro inesperado");
45.         return EXIT_FAILURE;
46.    }
47. return EXIT_SUCCESS;
48. }
```

O exemplo acima, apesar de simples, executa bem a função a que se destina. É um programa que soma dois inteiros, não negativos. O controle é feito através do uso da estrutura switch (linhas 29 a 45) – poderia ser usado if's também – que associa os valores de retorno de setjmp com o tipo de função disparada. (linhas 6 a 13)