

Зачем нужны функции setjmp longjmp?

Вопрос задан 5 лет 4 месяца назад Modified 5 лет 4 месяца назад Просмотрен 3к раз



5



2



Здравствуйте, Этот вопрос остается не освещенным в учебниках по программированию на языке C. Было бы интересно знать, зачем и как применяют функции **setjmp** и **longjmp**. Если возможно то приведите, какойнибудь, пример их использования. Спасибо.



Поделиться Улучшить вопрос Отслеживать

задан 18 ноя 2016 в 5:52



Вячеслав

582 3 16

3 ответа

Сортировка:

Highest score (default)



8



Функции `setjmp` и `longjmp` применяются почти для того же, для чего в новых языках придуманы конструкции `try/catch` и `throw`. Последние могут вызывать `setjmp` и `longjmp` в некоторых реализациях по крайней мере в C++

Рассмотрим такой случай вызова функциями в программе: `f1` вызывает `f2`, а та в свою очередь `f3`, которая вызывает себя 100 раз рекурсивно. В стек попадут параметры, адреса возврата и локальные переменные каждой запущенной функции. Всё это вместе называется *фреймом стека*, и в нашем случае в стек будут занесены 102 фрейма.

Адрес возврата — это физический адрес инструкции, которая должна быть выполнена следующей после завершения вызванной функции. Этот адрес сохраняется в стеке при выполнении машинной инструкции, которая называется что-то вроде `CALL` и восстанавливается при выполнении инструкции, которая называется что-то вроде `RET`.

Обычный способ вернуться в функцию `f1` из самой вложенной функции заключается в том, что все функции завершаются стандартным образом, и удаляют свои фреймы из стека по одному. Это, как понимаете, долго. В случае *исключительной ситуации* хотелось бы сразу попасть наверх, туда, где происходит глобальная обработка исключительных ситуаций.

Нормальным способом это сделать невозможно, но в старые времена программисты придумали довольно простой хак. Стек это просто память, а указатель на вершину стека — один из регистров процессора. При занесении в стек значений, всё, что там было раньше, остаётся неизменным, поэтому для того, чтобы восстановить стек, достаточно всего лишь вернуть значение указателя к тому, которое было, например, у функции `f1`.

Но, чтобы его вернуть, надо сначала его запомнить. На самом деле запомнить надо также и указатель текущей инструкции, потому что при возврате надо не только восстановить стек, но и передать управление на инструкцию внутри функции `f1`.

Для этого используется функция `setjmp`:

```
#define TOO_DEEP_RECURSION 1
jmp_buf env;

int main()
{
    int val;

    val = setjmp(env);
    if(val) {
        fprintf(stderr, "Возникла ошибка с кодом %d", val);
        return val;
    }

    recursive100(1);

    return 0;
}

recursive100(int level)
{
    if (level == 100)
        longjmp(env, TOO_DEEP_RECURSION);

    recursive100(level + 1);
}
```

Поскольку это хак, то код выглядит несколько странно, но надо понять принцип. При первом вызове `setjmp` функция сохраняет все необходимые указатели в структуре `jmp_buf` и возвращает 0. Как видите, в этом случае выполнение программы продолжается стандартным образом и она вызывает рекурсивно 100 функций.

При 100-м вызове мы возвращаемся назад с помощью `longjmp`. Эта функция получает на вход структуру, извлекает из неё указатель следующей инструкции и указатель на вершину стека, и восстанавливает их. Второй параметр это целое число, которое не должно быть равно 0.

Управление снова передаётся в `main` так, как будто только что произошёл возврат из `setjmp` и функция возвращает целое число, которое вы передали в `longjmp`. На этот раз оно не равно 0, функция `main` выводит сообщение об ошибке и останавливается.

Функций `longjmp` в вашей программе может быть несколько, каждая может возвращать свой код и на основе кода вы можете выводить разные сообщения.

Естественно, вы можете создать несколько переменных `jmp_buf`, и организовать сложную логику возвратов. Важно помнить две вещи:

1. Перед вызовом `longjmp` структура `jmp_buf` должна быть проинициализирована через `setjmp`, иначе поведение программы будет непредсказуемым.

2. При быстром возврате чистится только стек. Не выполняется никакой код освобождения других ресурсов, например, освобождения памяти из кучи, закрытие файлов, закрытие сокетов и прочее.

Поделиться Улучшить ответ

изменён 18 ноя 2016 в 9:54

ответ дан 18 ноя 2016 в 7:30

Отслеживать



Mark Shevchenko

11.4k 17 37

Спасибо, понятно написано. Но нужно быть осторожным, что бы не передать управление в несуществующий контекст. – Вячеслав 18 ноя 2016 в 8:41

@Вячеслав Да, конечно, longjmp без предварительного setjmp гарантированно уронит программу. – Mark Shevchenko 18 ноя 2016 в 9:47



3



Пример использования longjmp из моей реальной жизни. Делал достаточно объёмную утилиту для тестирования железяки, которую мы производим. Там, в частности, был тест памяти. Требовалось, что бы тестирование выполнялось до нажатия Ctrl/C, после чего выполнялся переход на следующий этап тестирования. Делал так:

```
// Задаём реакцию на нажатие комбинации клавиш Ctrl/C
signal(SIGINT, sig_sigint);
. . .
// Перед началом цикла запоминаем положение:
if (setjmp(jmpbuffer) != 0) {
    // Длинный переход из обработчика сигнала завершения
    printf("\n\nПолучен сигнал завершения работы\n");
    goto end_loop;
}
// Сам цикл тестирования
do {
    . . .
} while (do_work !=0);
end_loop;
```

Таких этапов тестирования было несколько. И после каждого нажатия Ctrl/C программа переходила к СЛЕДУЮЩЕМУ этапу. Попробуйте сделать такое на try/catch! :-)) И не надо говорить про оператор goto! В данном контексте, с переходом вперёд, его использование абсолютно уместно.

Поделиться Улучшить ответ Отслеживать

ответ дан 18 ноя 2016 в 10:42



Sergey

12.7k 11 24



Мой вариант:

0



```
typedef enum {
    NoException = 0,
    Exception,
    ArgumentException,
    NotImplementedException,
    /* ... */
};
```

```
} __exception_types;

#define try          /* ... */
#define catch(x)     /* ... */
#define throw(x,...) /* ... */
#define finally      /* ... */
```

Использование:

```
try {
    puts("throw(NotImplementedException) call...");
    throw(NotImplementedException, "in future releases only");
}
catch( NotImplementedException ) {
    printf("catch NotImplementedException" );
    if( __exception_with_msg ) {
        printf( ": %s", __exception_msg );
    }
    printf( "\n" );
}
else {
    printf("exception %d", __exception_type );
    if( __exception_with_msg ) {
        printf( ": %s", __exception_msg );
    }
    printf( "\n" );
}
finally {
    puts( "finally() block" );
}
```

Содержимое дефайнов предлагаю придумать самостоятельно :)

Поделиться Улучшить ответ

изменён 18 ноя 2016 в 6:49

ответ дан 18 ноя 2016 в 6:27

Отслеживать



PinkTux

9,036 12 26

Видимо все так и делают. Но это не так просто как в ++! – Вячеслав 18 ноя 2016 в 6:55

@Вячеслав, кто эти все? Когда придумывал свой вариант нашёл всего две, ну две с половиной, приличных реализации :) Думаю, потому что на самом деле это особо и не нужно никому. Мне тоже :) – PinkTux 18 ноя 2016 в 7:00

Профессионалы... – Вячеслав 18 ноя 2016 в 8:18

1 @Вячеслав, область применения у них всегда одна: возврат к какому-то состоянию с восстановлением контекста. [Например](#). И примеры уже показали, чем они не нравятся? Или они для вас недостаточно "классически"? Так уверяю, принципиально других примеров и не будет. Кстати, что вы подразумеваете под "сопрограммами" и почему считаете их, наравне с генераторами, архаичной ерундой? – PinkTux 18 ноя 2016 в 8:50 ✎

1 @Вячеслав, вы уж сами определяйтесь, нужны вам сопрограммы или это архаика :-)) – PinkTux 18 ноя 2016 в 9:47
