

Рекурсия

Любой современный язык программирования позволяет создать функцию — участок кода, которому присваивается идентификатор, и который может быть вызван на выполнение из другого участка кода.

При этом нет фундаментальных ограничений на то место, откуда можно вызвать функцию. В том числе, можно вызвать функцию изнутри нее же.

Рекурсия — это ситуация, при которой функция вызывает сама себя.

Различают **прямую** рекурсию, когда функция вызывает себя непосредственно, и **косвенную** рекурсию, когда функция вызывает другую функцию, а она в свою очередь вызывает исходную.

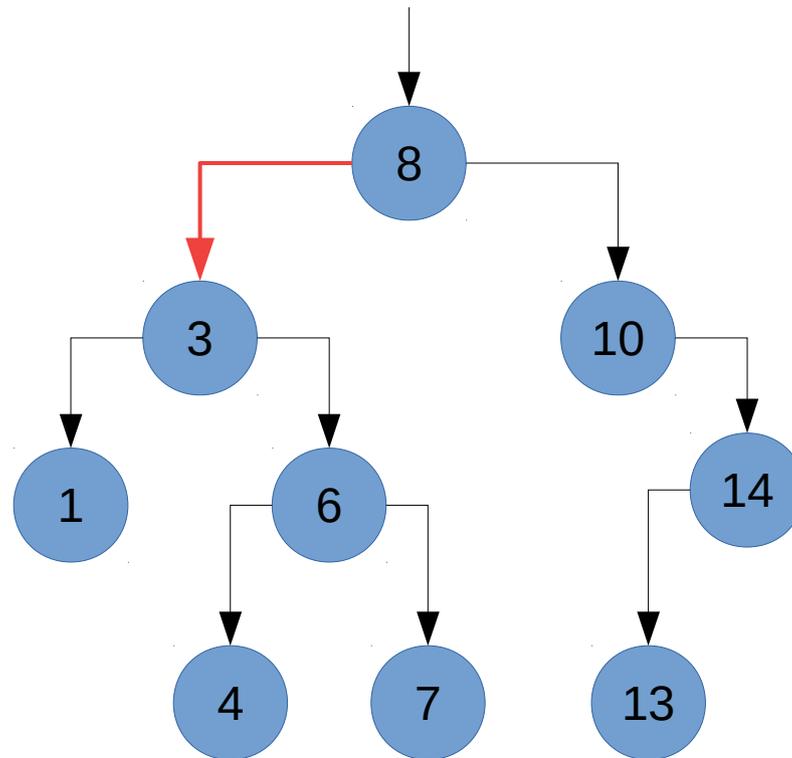
Рекурсия

Рекурсия используется для реализации рекуррентных алгоритмов, а также для обработки данных, хранящихся в виде дерева (например, файлов).

Функция, осуществляющая обход дерева, будет выполнять одни и те же операции независимо от того, какой узел дерева был выбран в качестве начального. При переходе к следующему узлу, функция вызывает себя же, передав новый узел в качестве аргумента.

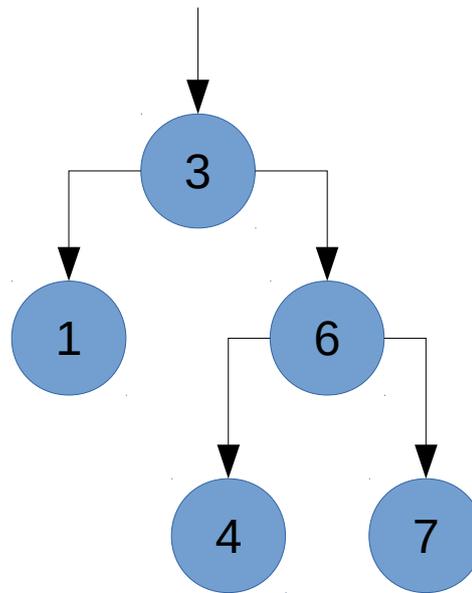
Рекурсия

Очевидно, что ситуация, когда функция вызывает сама себя, может привести к закливанию программы. Поэтому обязательно необходимо определить условие, задающее **терминальную ветвь**, ограничивающую рекурсию. Ветви расчета, ведущие к продолжению рекурсии, называются **рекурсивными**.



Рекурсия

Очевидно, что ситуация, когда функция вызывает сама себя, может привести к закливанию программы. Поэтому обязательно необходимо определить условие, задающее **терминальную ветвь**, ограничивающую рекурсию. Ветви расчета, ведущие к продолжению рекурсии, называются **рекурсивными**.



Пример рекурсии

Один из примеров применения рекурсии — расчет факториала. Можно записать соотношение:

$$n! = n \cdot (n-1)!$$

И, исходя из него записать функцию расчета факториала следующим образом:

```
def fact(n):  
    if n == 1:  
        return 1  
    return n * fact(n-1)
```

Условие `n == 1` определяет терминальную ветвь рекурсии. Остальная часть функции является рекурсивной ветвью.

Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

```
f = fact(3)
```

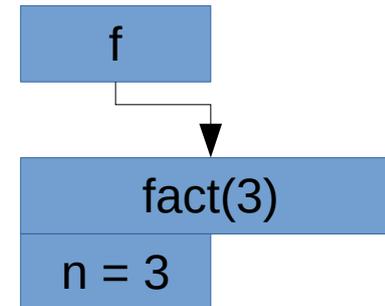
f

Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r
```

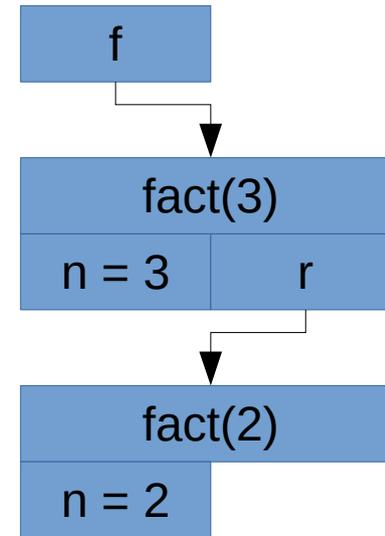
```
f = fact(3)
```



Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

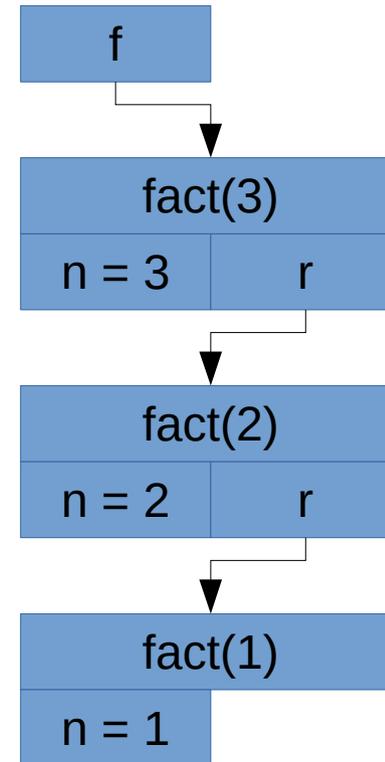
```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r  
  
f = fact(3)
```



Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

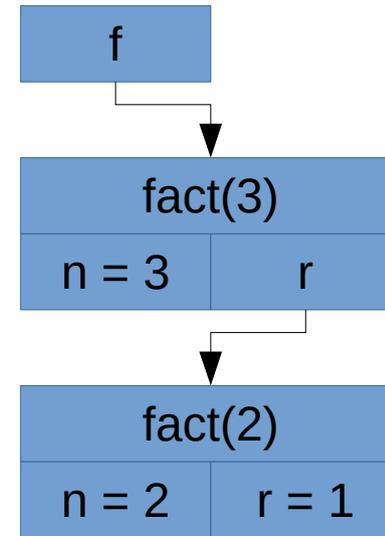
```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r  
  
f = fact(3)
```



Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r  
  
f = fact(3)
```

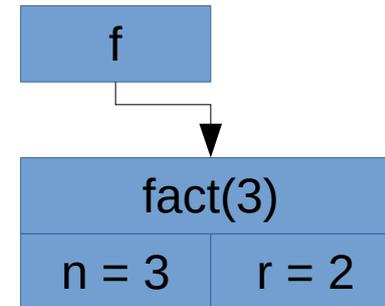


Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r
```

```
f = fact(3)
```



Пример рекурсии

Каждый вызов функции работает со своим набором аргументов и своими локальными переменными

```
def fact(n):  
    if n == 1:  
        return 1  
    r = n * fact(n-1)  
    return r
```

```
f = fact(3)
```

f = 6

Стек вызовов

Каждый вызов функции несет накладные расходы: передачу аргументов, хранение возвращаемого значения и адреса возврата, с которого следует продолжить выполнение по окончании работы функции.

Для хранения этих данных используется специальная область памяти — **стек**. Туда же помещаются локальные переменные, создаваемые внутри функции. При этом объем стека ограничен величинами порядка единиц мегабайт.

При использовании рекурсии возможно возникновение нетривиальной ошибки — **переполнение стека** (англ. Stack Overflow).

Аппаратный стек

Откомпилированная программа, исполняемая непосредственно на центральном процессоре, использует аппаратный стек.

Стек обслуживается одним регистром процессора, хранящим адрес вершины стека. На процессорах семейства Intel он носит название ESP (32 бита) и RSP (64 бита).

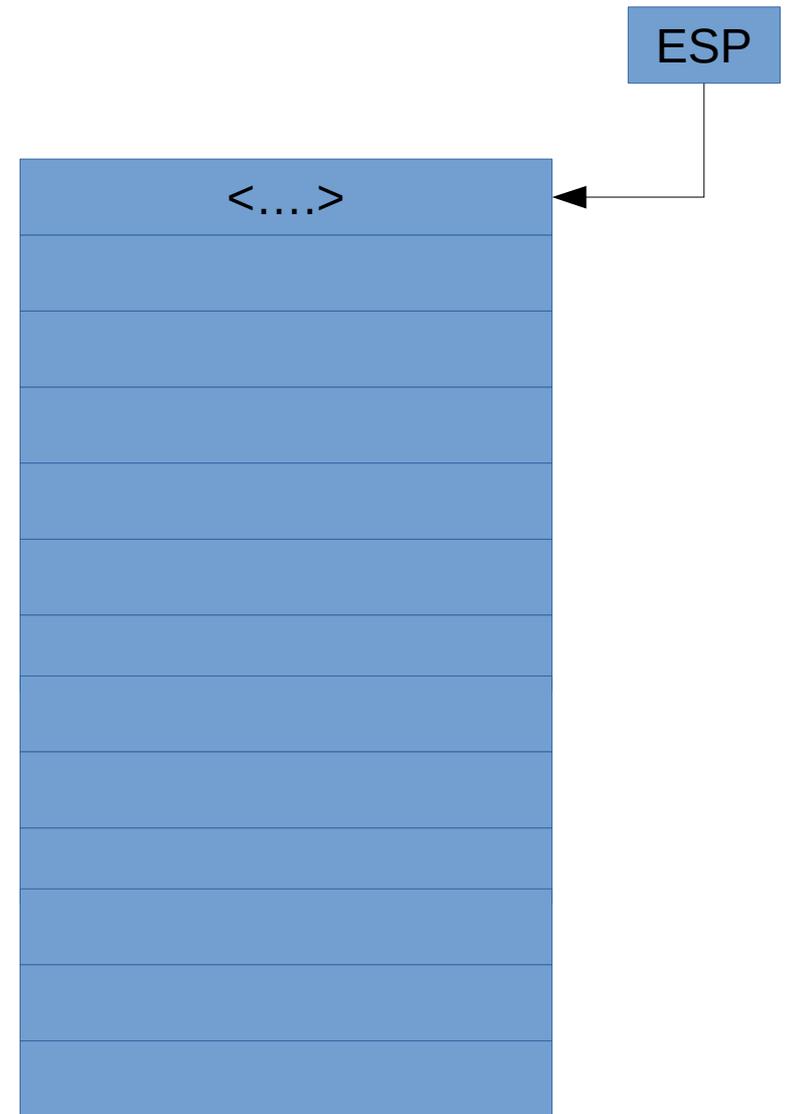
Добавление данных в стек происходит путем изменения адреса вершины стека и копирования данных по этому адресу.

Удаление данных происходит простым изменением адреса обратно.

После окончания вызова функции адрес вершины стека должен вернуться в то же значение, которое было до вызова функции. Работу со стеком полностью берет на себя компилятор.

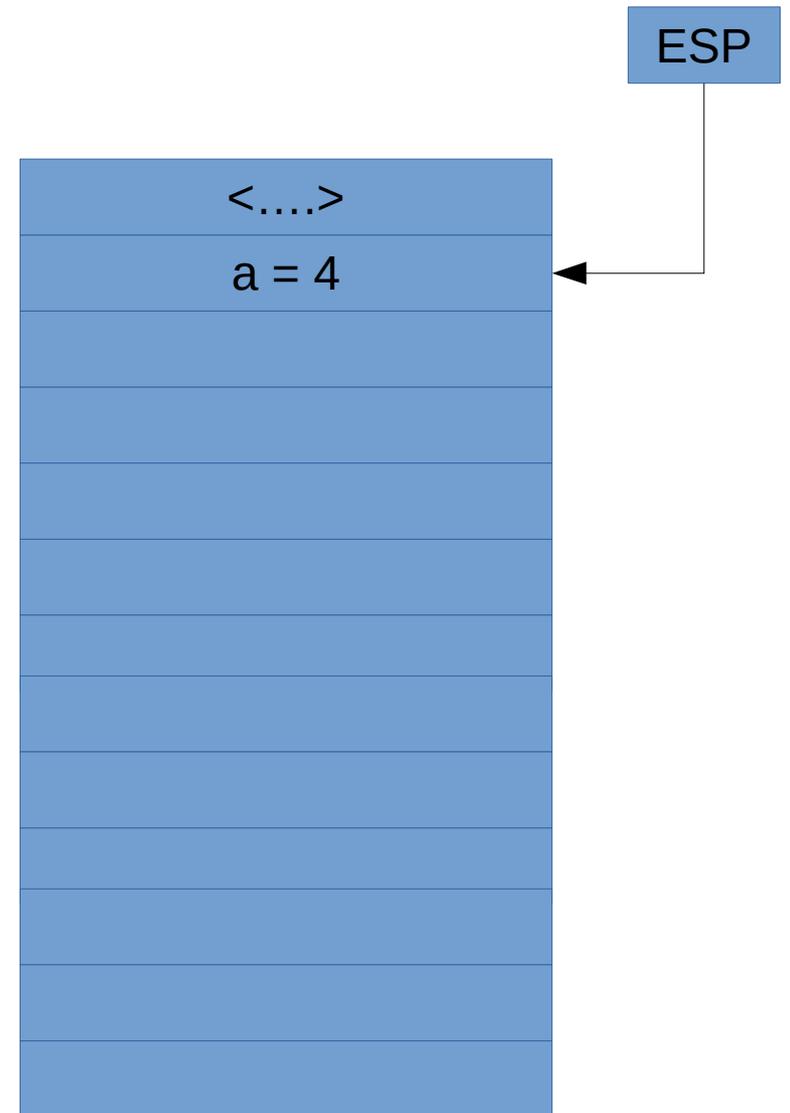
Аппаратный стек

```
20: int main()  
21: {  
22:     int a = 4;  
23:     int b;  
24:     b = pow(a, 2);  
25:     printf("%d\n", b);  
26:     return 0;  
27: }
```



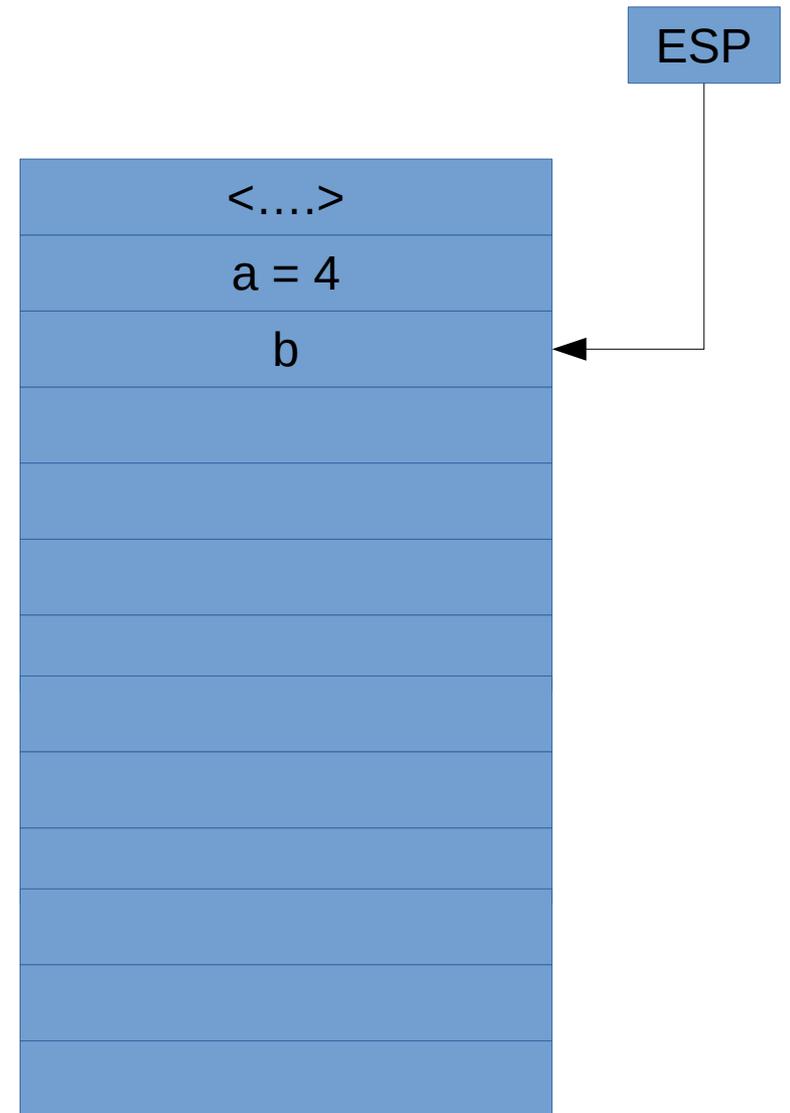
Аппаратный стек

```
20: int main()
21: {
22:     int a = 4;
23:     int b;
24:     b = pow(a, 2);
25:     printf("%d\n", b);
26:     return 0;
27: }
```



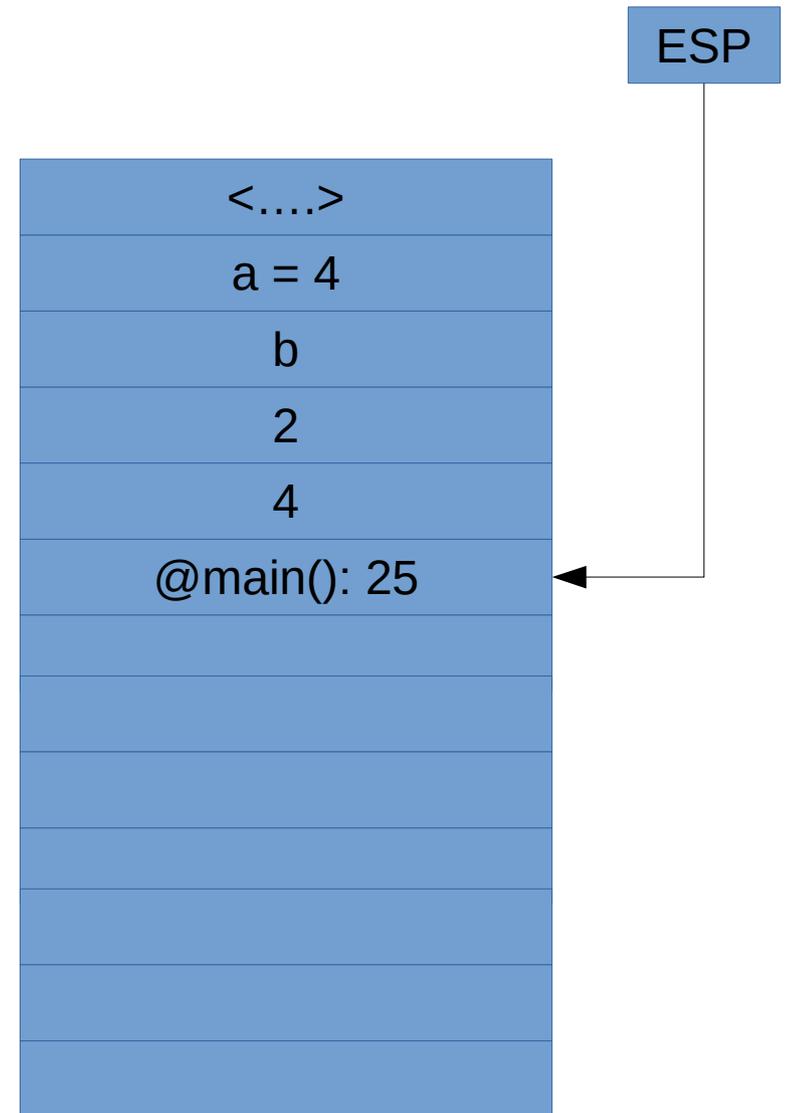
Аппаратный стек

```
20: int main()
21: {
22:     int a = 4;
23:     int b;
24:     b = pow(a, 2);
25:     printf("%d\n", b);
26:     return 0;
27: }
```



Аппаратный стек

```
20: int main()
21: {
22:     int a = 4;
23:     int b;
24:     b = pow(a, 2);
25:     printf("%d\n", b);
26:     return 0;
27: }
```



Аппаратный стек

```
20: int main()
21: {
22:     int a = 4;
23:     int b;
24:     b = pow(a, 2);
25:     printf("%d\n", b);
26:     return 0;
27: }
```

EAX=16

ESP

<....>

a = 4

b = 16

2

4

@main(): 25

Аппаратный стек

```
20: int main()
21: {
22:     int a = 4;
23:     int b;
24:     b = pow(a, 2);
25:     printf("%d\n", b);
26:     return 0;
27: }
```

EAX=16

ESP

<....>

a = 4

b = 16

"%d\n"

16

@main(): 26

Стек вызовов

Отладчик работает в паре с компилятором и знает, в каком формате в аппаратном стеке хранятся аргументы и адреса возврата.

Это позволяет отладчику на основе информации в стеке составить **стек вызовов**. Он показывает последовательность вызова функций и передаваемые им аргументы, при помощи которых программа дошла до текущего места выполнения.

Стек вызовов доступен при постановке программы на паузу отладчиком.

Постановка на паузу может быть сделана вручную во время выполнения, либо автоматически при достижении определенной строки (breakpoint).

Стек вызовов

The screenshot displays the Qt Creator IDE interface. The main editor shows the implementation of the `mazestep` function in `maze.cpp`. The function signature is `void mazestep(QVector<QByteArray>& a, int *rows, int *cols, int r, int c)`. The code defines a `vector` of size 3x2 and enters a `while(true)` loop. Inside the loop, it checks four directions (up, down, left, right) for valid moves. The current line of execution is at line 33, where `int i = 0; /* look around */` is being executed.

The Variable Watch window on the right shows the following data:

Имя	Значение	Тип
a	<21 элемент>	QVector<QByteArray> &
c	1	int
cc	9162188	int
cols	10	int
i	9162176	int
r	1	int
rows	10	int
rr	561454112	int
vector	@0x8bcd2c	int[3][2]

The Call Stack window at the bottom shows the current call stack:

Уровень	Функция	Файл	Строка	Номер	Функция	Файл	Строка	Адрес
1	mazestep	maze.cpp	33	3	...t *, int *, int, int)	...maze\maze.cpp	33	0x401fc1
2	mazewalk	maze.cpp	95					
3	MainWindow::createMaze	mainwindow.cpp	42					
4	MainWindow::qt_static_metacall	moc_mainwindow.cpp	74					
5	QObject::activate	qobject.cpp	3766					
6	QObject::activate	qobject.cpp	3628					
7	QAbstractButton::clicked	moc_qabstractbutton.cpp	308					
8	QAbstractButtonPrivate::emitClicked	qabstractbutton.cpp	414					
9	QAbstractButtonPrivate::click	qabstractbutton.cpp	407					
10	QAbstractButton::mousePressEvent	qabstractbutton.cpp	1011					
11	QWidget::event	QWidget.cpp	8817					
12	QAbstractButton::event	qabstractbutton.cpp	968					
13	QPushButton::event	QPushButton.cpp	682					
14	QApplicationPrivate::notify_helper	QApplication.cpp	3732					
15	QApplication::notify	QApplication.cpp	3208					
16	QCoreApplication::notifyInternal2	QCoreApplication.cpp	1044					
17	QCoreApplication::sendSpontaneousEvent	QCoreApplication.h	237					
18	QCoreApplicationPrivate::sendMouseEvent	QCoreApplication.cpp	2709					

Стек вызовов

The screenshot shows the Qt Creator IDE with the following components:

- Code Editor:** Displays the `mazewalk` function in `maze.cpp`. The function signature is `void mazewalk(QVector<QByteArray>& maze, int rows, int cols)`. The code includes comments and logic for setting maze boundaries and stepping.
- Stack of Calls Window:** Located on the right, it shows the current call stack:

Имя	Значение	Тип
cols	10	int
maze	<21 элемент>	QVector<QByteArray> &
rows	10	int
stcol	1	int
strow	1	int
- Debugger Console:** Shows the execution flow with the following stack of calls:

Уровень	Функция	Файл	Строка	Номер	Функция	Файл	Строка	Адрес
1	mazestep	maze.cpp	33	3	...t *, int *, int, int)	...maze/maze.cpp	33	0x401fc1
2	mazewalk	maze.cpp	95					
3	MainWindow::createMaze	mainwindow.cpp	42					
4	MainWindow::qt_static_metacall	moc_mainwindow.cpp	74					
5	QObject::activate	qobject.cpp	3766					
6	QObject::activate	qobject.cpp	3628					
7	QPushButton::clicked	moc_qabstractbutton.cpp	308					
8	QPushButtonPrivate::emitClicked	qabstractbutton.cpp	414					
9	QPushButtonPrivate::click	qabstractbutton.cpp	407					
10	QPushButton::mousePressEvent	qabstractbutton.cpp	1011					
11	QWidget::event	QWidget.cpp	8817					
12	QPushButton::event	qabstractbutton.cpp	968					
13	QPushButton::event	QPushButton.cpp	682					
14	QApplicationPrivate::notify_helper	QApplication.cpp	3732					
15	QApplication::notify	QApplication.cpp	3208					
16	QCoreApplication::notifyInternal2	QCoreApplication.cpp	1044					
17	QCoreApplication::sendSpontaneousEvent	QCoreApplication.h	237					
18	QCoreApplicationPrivate::sendMouseEvent	QCoreApplication.cpp	2709					

Программный стек

Программе, исполняемой интерпретатором, аппаратный стек недоступен. Вместо этого интерпретатор организует свой программный стек.

Вместо программных адресов используются номера строк в сценарии, к выполнению которых следует вернуться после завершения функции, а также свои соглашения о хранении аргументов и возвращаемых значений.

Программный стек может быть получен даже в процессе выполнения.

Также можно штатно обработать ошибку переполнения стека вызовов на уровне интерпретатора.

Программный стек

Пример иллюстрирует сценарий, который вызывает рекурсивно одну и ту же функцию до переполнения стека и ловит данную ошибку при помощи блока try ... except:

```
def rec(i):
    print("Recursion depth="+str(i))
    try:
        rec(i+1)
    except RuntimeError as re:
        print(re.args)

rec(0)
```

Результат запуска сценария:

```
<...>
Recursion depth=995
Recursion depth=996
('maximum recursion depth exceeded while calling a Python object',)
```

Применение рекурсии

- Обработка данных, сформированных в виде дерева (например, поиск файла в файловой системе).
- Алгоритмы обработки массивов, разделяющие массивы на части (например, быстрая сортировка).
- Расчет рекуррентных задач, в которых следующий шаг алгоритма равен предыдущему, но отличается начальными значениями.