

变量的作用域

huiw@suda.edu.cn

```
int sum_digits(int n) {
```

```
    int sum = 0; /* local variable */
```

```
    while (n > 0) {  
        sum += n % 10;  
        n /= 10;  
    }
```

```
    return sum;
```

```
}
```

形式参数

局部变量(local variable)

```
int main (int argc, char* argv[]) {  
    return sum_digits(sum_digits(123));  
}
```

while语句

```
iteration_statement  
    : while '(' logic expression ')' statement
```

递归版本

```
typedef unsigned int u32;

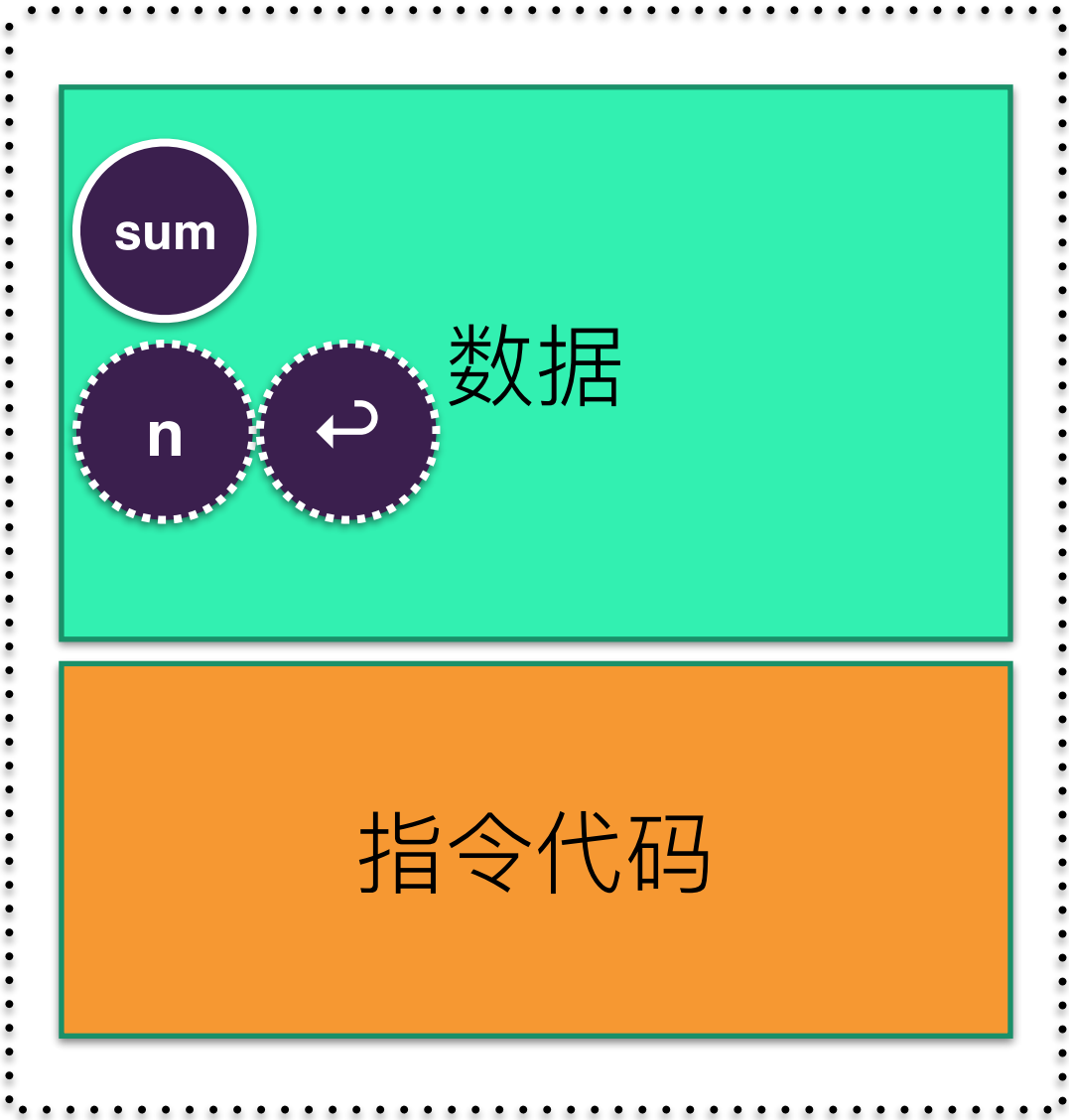
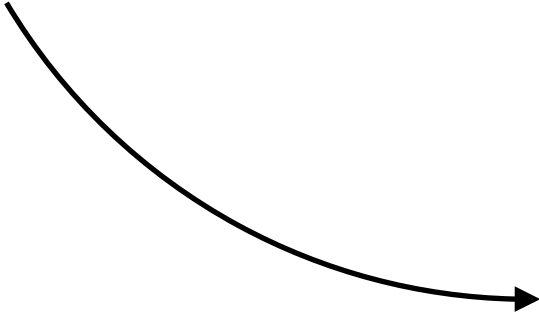
u32 sum_digits(u32 n) {
    if (n < 10) return n;

    u32 sum = n % 10;

    return sum + sum_digits(n / 10);
}

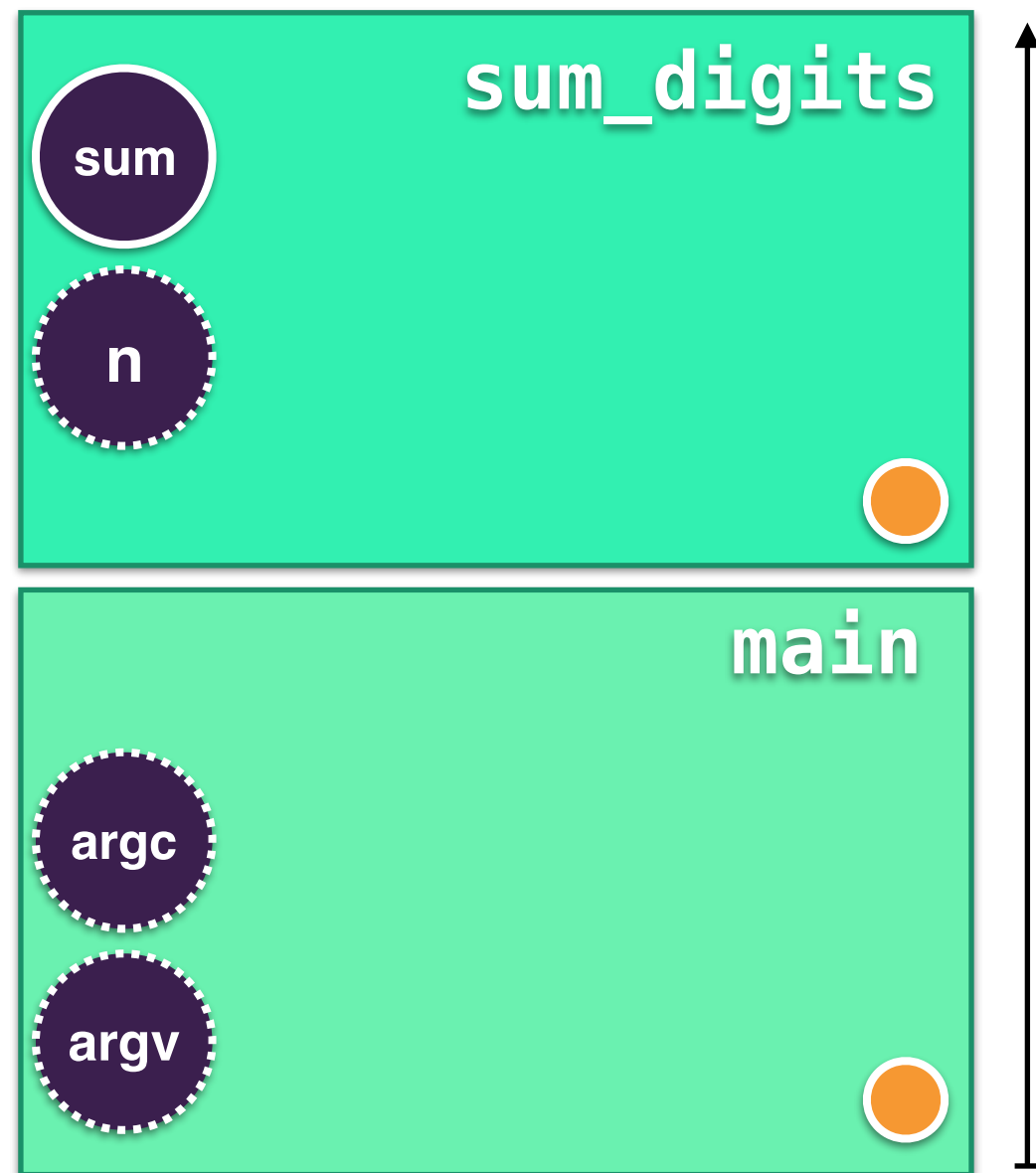
int main (int argc, char* argv[]) {
    return sum_digits(12);
}
```

sum_digits



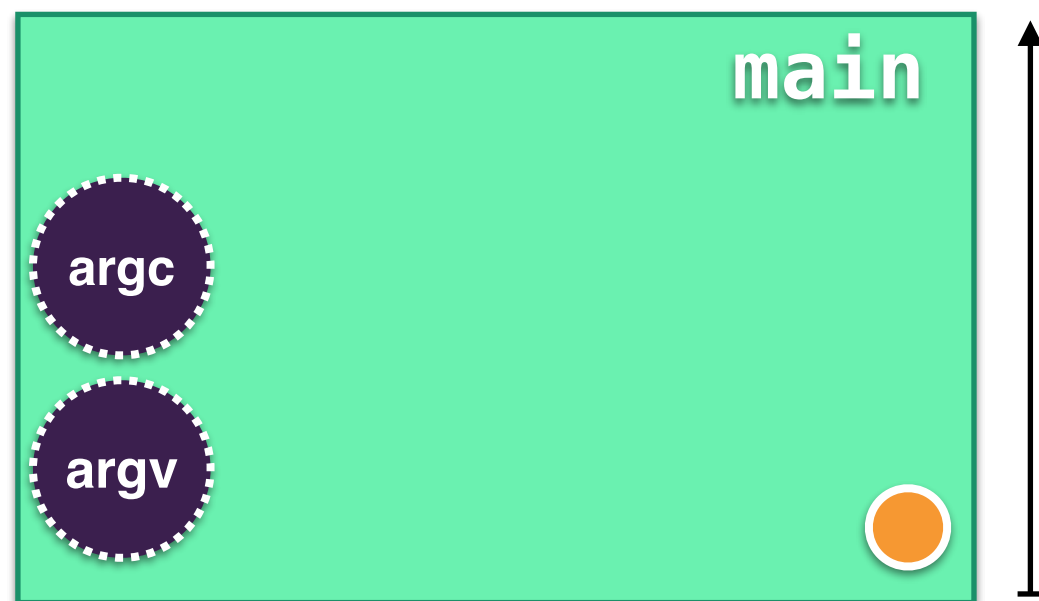
当函数被调用的时候，
才会在栈上为使用到的变量分配存储区域。

调用栈



调用结束，**主调函数**记录返回值。
原先为**被调函数**分配的栈上存储区域，出栈释放。

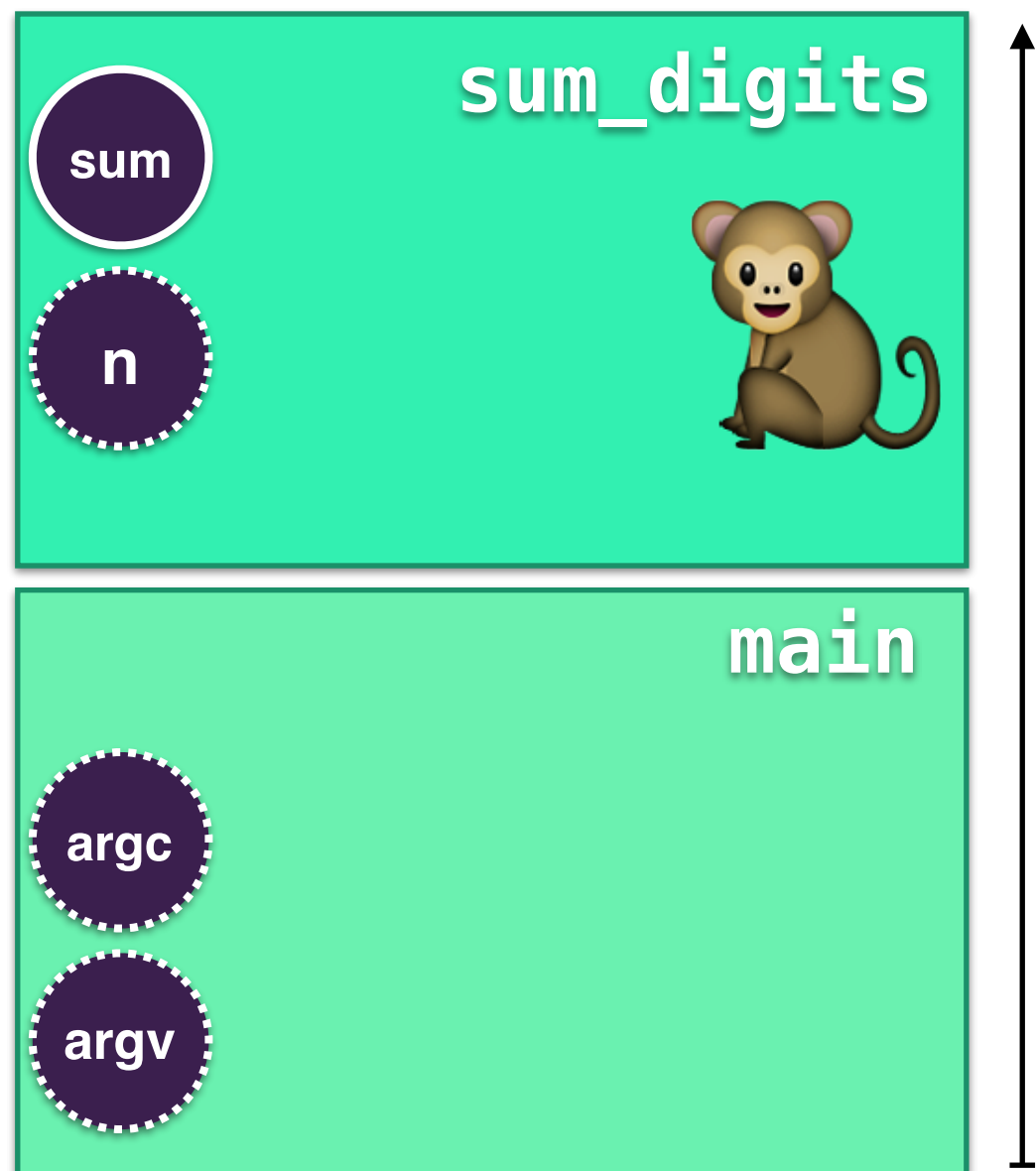
调用栈



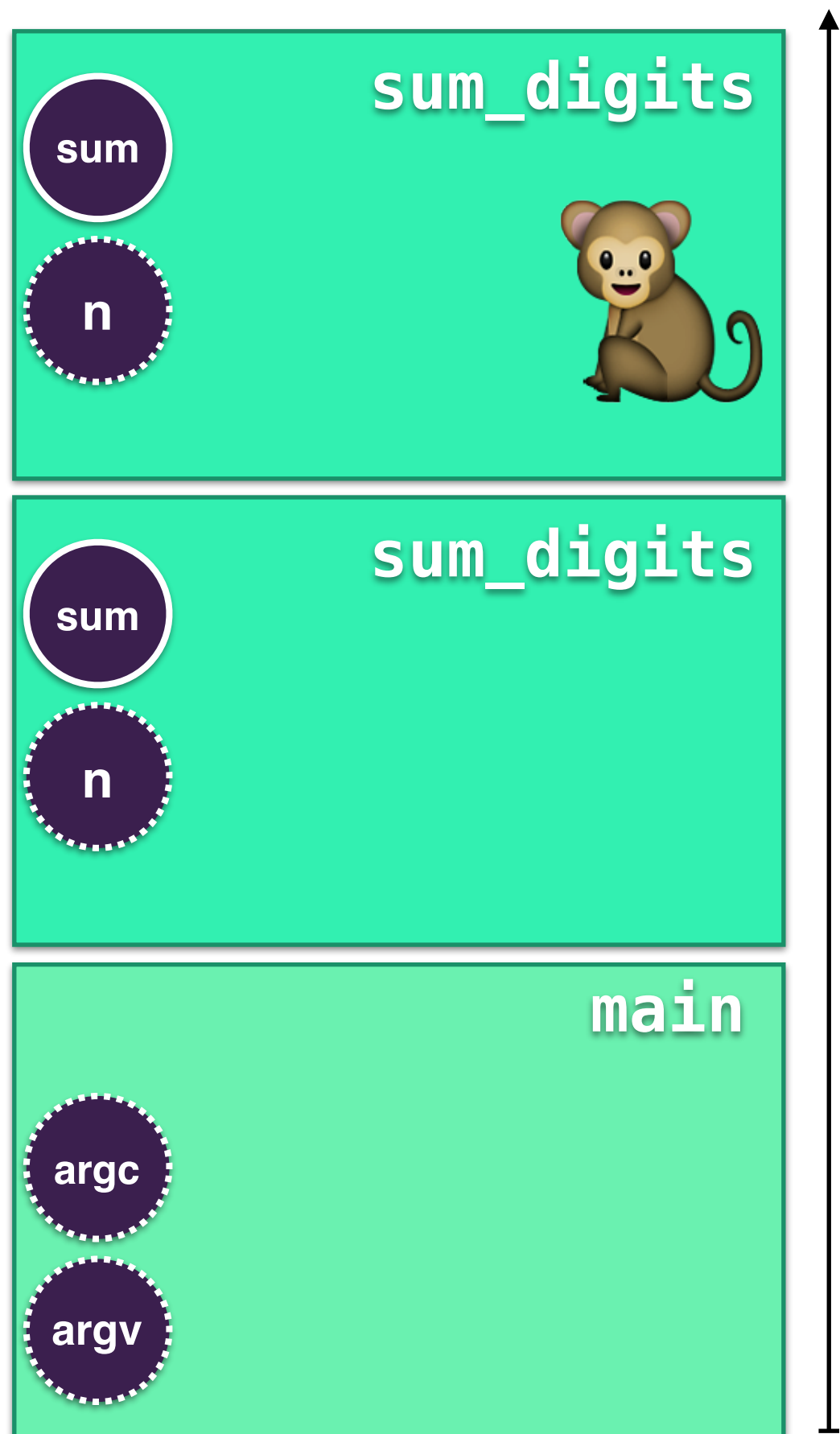
显然，函数中的局部变量是有生命期的。

调用开始，出生并存活
调用结束，移交并死亡

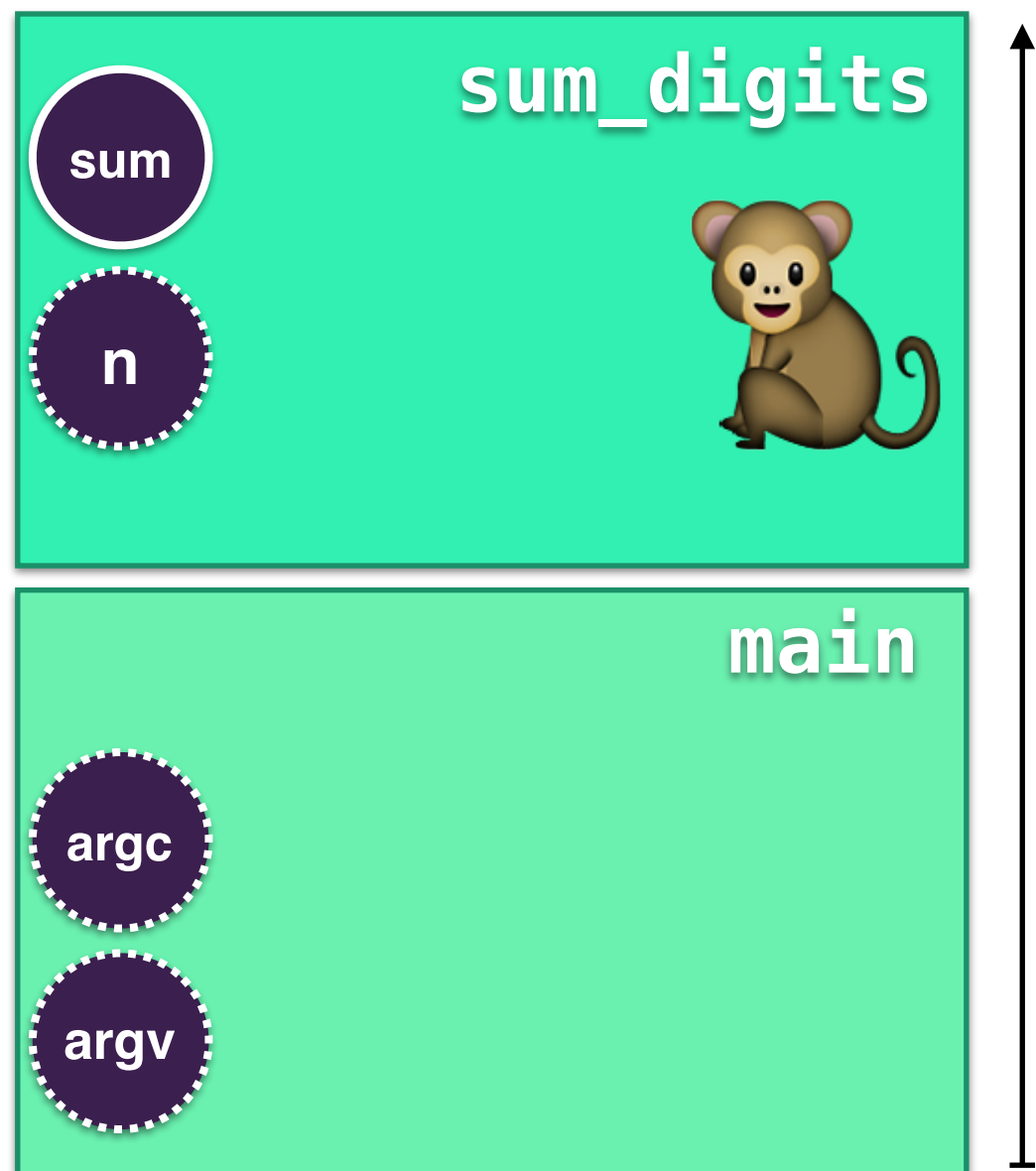
调用栈



调用栈

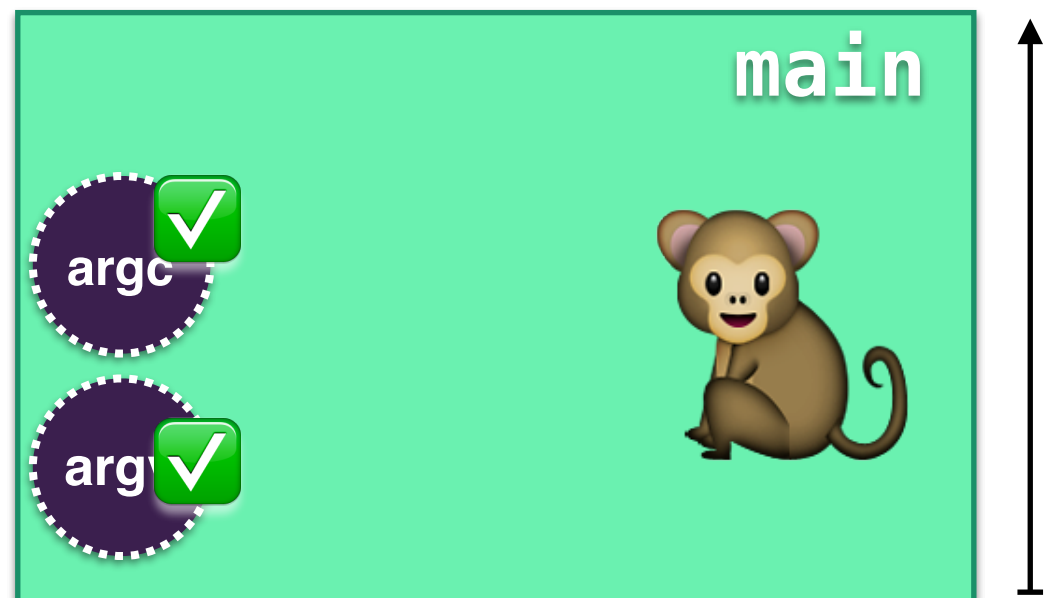


调用栈



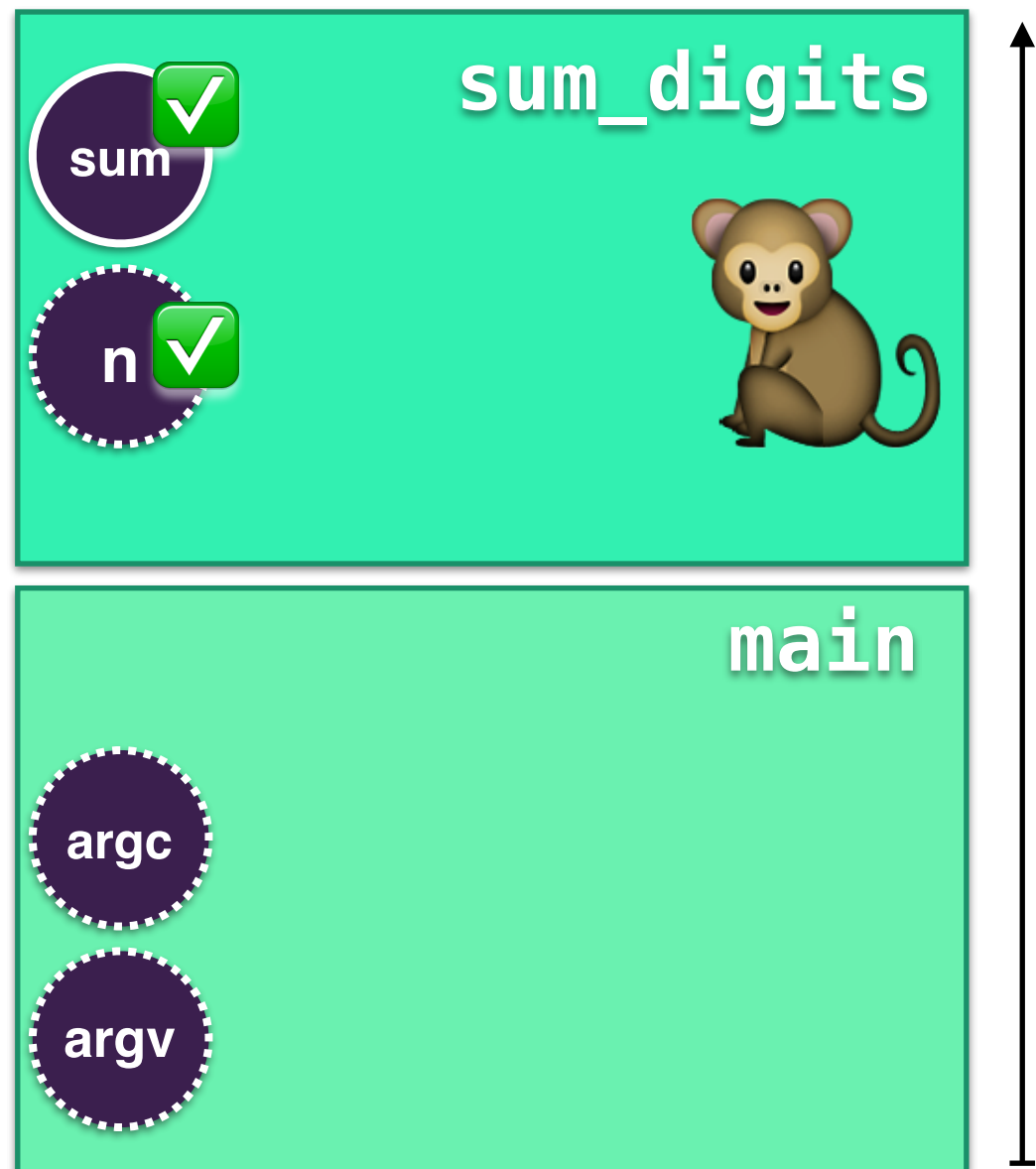
小猴子能够看到的变量有那些？

调用栈

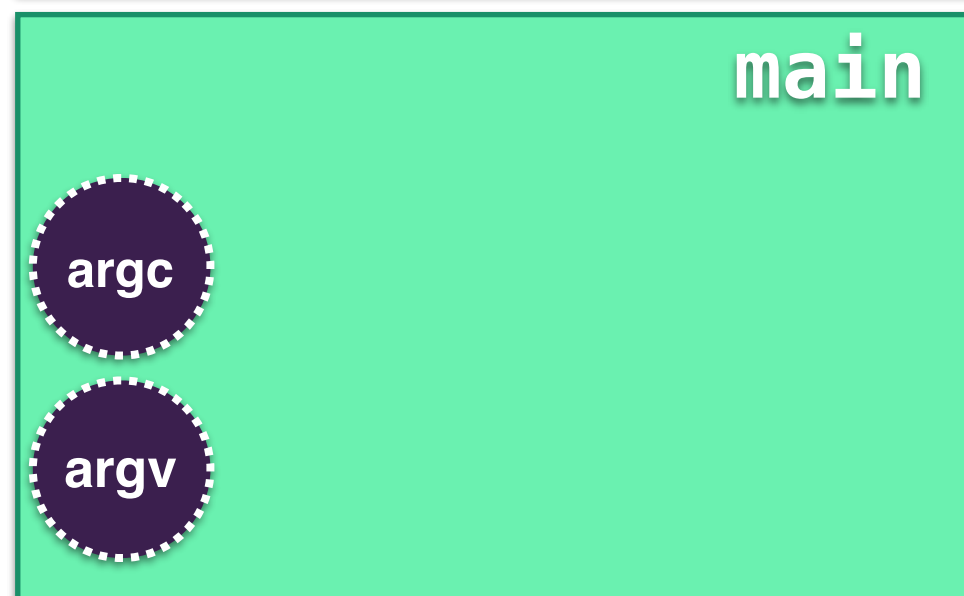
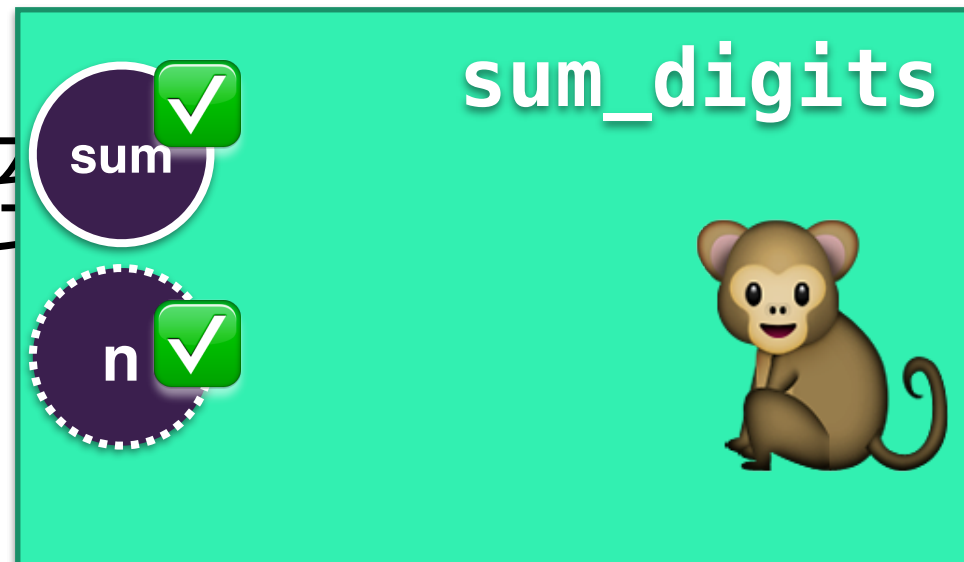


小猴子能够看到的变量有那些?

调用栈



小猴子能够看到



调用栈

小猴子能够看到的变量有那些?

调用栈



小猴子能够看到的变量有那些？

调用栈



变量的作用域

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

```
int global_variable = 10;
```

```
int square(int x)
{
    return x * x;
}
```

```
int main (int argc, char* argv[]) {
```

```
    int sum = 0;
```

```
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++){
            printf("%2d * %2d = %2d", i, j, i*j);
        }
    }
```

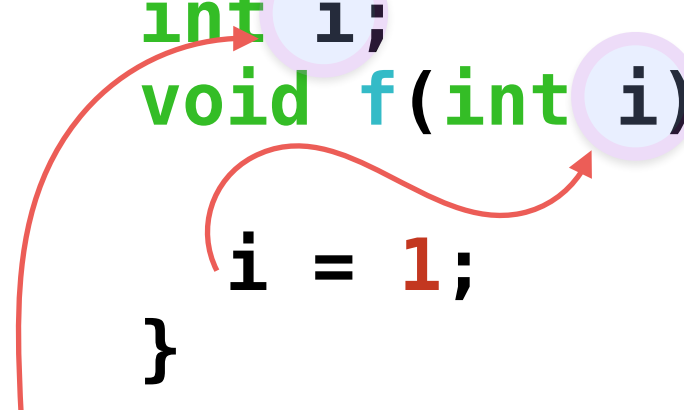
```
    sum += i;
```

```
}
```

```
    return sum;
```

```
}
```

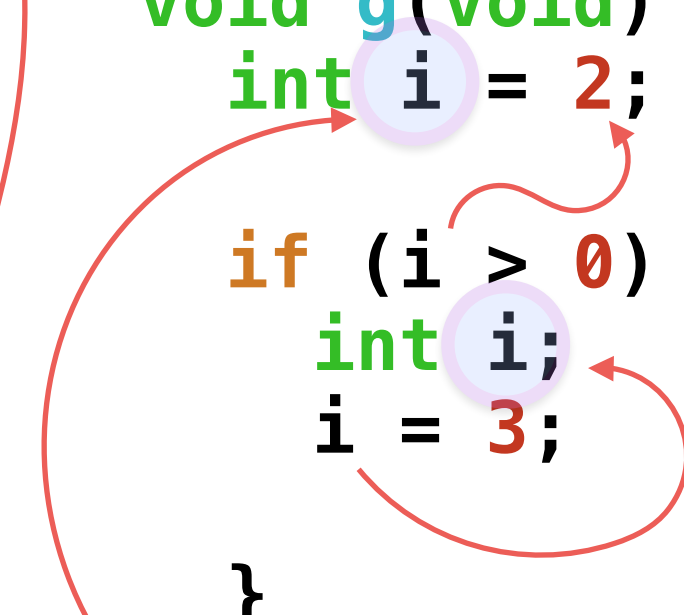
```
int i;  
void f(int i) {  
    i = 1;  
}
```



A red arrow originates from the parameter `i` in the function signature `void f(int i)` and points to the global variable `i` declared at the top of the scope.

```
/* declaration 1 */  
/* declaration 2 */
```

```
void g(void) {  
    int i = 2;  
    if (i > 0) {  
        int i;  
        i = 3;  
    }  
    i = 4;  
}
```



A red arrow originates from the parameter `i` in the function signature `void g(void)` and points to the local variable `i` declared inside the function body, specifically to the `int i = 2;` line.

```
/* declaration 3 */
```

```
/* declaration 4 */.
```

```
void h(void) {  
    i = 5;  
}
```

全局变量有专门的存储区域。

```
static int j;

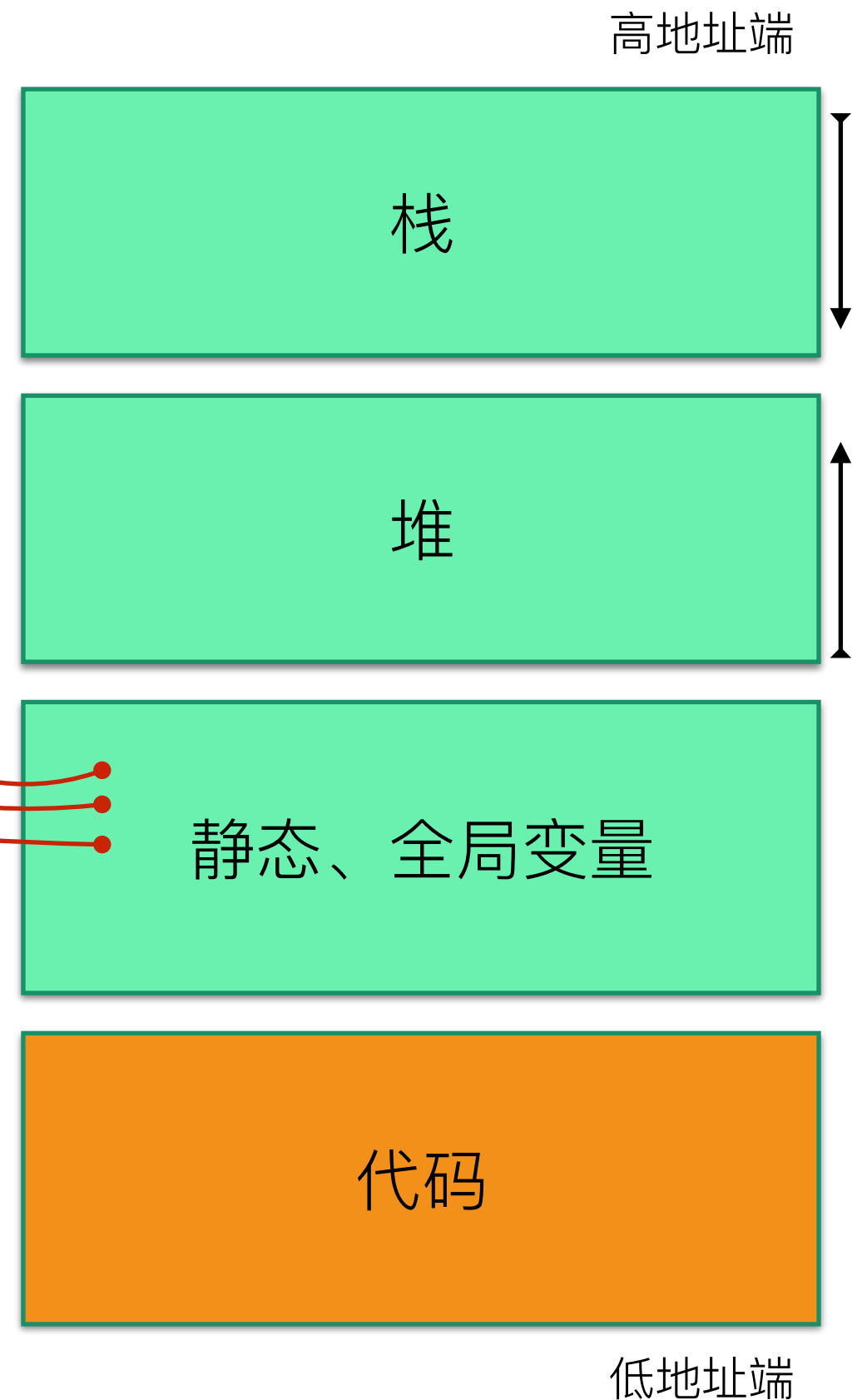
void fun1(void) {
    static int i = 0;
    i++;
}

void fun2(void) {
    j = 0;
    j++;
}
int global_var;

int main (int argc, char* argv[]) {

    for (int k = 0; k < 10; k++) {
        fun1();
        fun2();
    }
    return 0;
}
```

static_var.c



static_var.c

```
static int j;  
  
void fun1(void) {  
    static int i = 0;  
    i++;  
}  
  
void fun2(void) {  
    j = 0;  
    j++;  
}  
int global_var;  
  
int main (int argc, char* v[]  
  
    for (int k = 0; k < 10; ) {  
        fun1();  
        fun2();  
    }  
    return 0;  
}
```

```
static int j;
```

```
void fun1(void) {
```

```
    extern int global_var_here;  
    global_var_here += 20;
```

```
    static int i = 0;  
    i++;
```

```
}
```

```
void fun2(void) {
```

```
    j = 0;  
    j++;
```

```
}
```

```
int global_var_here = 18;
```

```
int main (int argc, char* argv[]) {
```

```
    for (int k = 0; k < 10; k++) {  
        fun1();  
        fun2();  
    }
```

```
    return 0;
```

```
}
```

声明外部变量

声明定义全局变量

extern_var.c

```
static int j;  
  
void fun1(void) {  
    extern int global_var_here;  
    global_var_here += 20;  
  
    static int i = 0;  
    i++;  
}  
  
void fun2(void) {  
    j = 0;  
    j++;  
}
```

main.c

```
void fun1(void);  
void fun2(void);  
  
int global_var_here = 18;  
  
int main (int argc, char* argv[]) {  
    for (int k = 0; k < 10; k++) {  
        fun1();  
        fun2();  
    }  
    return 0;  
}
```

穿越文件使用全局变量

对于全局变量和静态变量，
在声明定义完成后，都会初始化为0.

指定变量的存储位置

auto	int a;	栈
	int b;	栈
register	int c;	寄存器
static	int d;	全局/静态变量存储区

使用静态变量

```
void
move(char from_pole, char to_pole) {
    static unsigned int counter = 0;
    counter++;
    printf("%4u: %c -> %c\n", counter, from_pole, to_pole);
}
```

```
void.
hanoi_solver(uint32_t n, char from_pole, char accessory_pole, char to_pole) {

    if (1==n) {
        move(from_pole, to_pole);
        return;
    }

    hanoi_solver(n-1, from_pole, to_pole, accessory_pole);
    move(from_pole, to_pole);
    hanoi_solver(n-1, accessory_pole, from_pole, to_pole);

}
```

共享使用全局变量的函数之间会形成隐形的依赖约束关系。

所以，要谨慎使用全局变量。

实现一个整数栈

```
#include <assert.h>
#include <stdbool.h> /* C99 */

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void) { top = 0; }
bool is_empty(void) { return top == 0; }

bool is_full(void) { return top == STACK_SIZE; }

void push(int i) {
    assert(!is_full());
    contents[top++] = i;
}

int pop(void) {
    assert(!is_empty());
    return contents[--top];
}
```

想一想， 对于一个使用栈的编程者，
他需要知道什么样的接口呢？

提示： 接口包括对数据对象集合的描述 + 一组函数声明， 在满足功能要求的同时， 越简单越好。

```
void make_empty(void);
```

```
int pop(void);
```

```
void push(int i);
```

括号匹配判断

((()()()((())))



(((((())()))))



```
#include <assert.h>
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
```

```
#define ERR_BAD_CHAR -1
```

```
void make_empty(void);
void push(int i);
int pop(void);
bool is_empty(void);
bool is_full(void);
```



```
bool
paired_parentheses(char a[], size_t sz) {

    make_empty();
    for (int i=0; i<sz; i++)
    {
        assert(a[i] == '(' || a[i] == ')');

        if (a[i] == '(')
            push('(');

        if (a[i] == ')')
            pop();
    }

    return is_empty();
}
```

```
bool
paired_parentheses_switch(char a[], size_t sz) {

    make_empty();
    for (int i=0; i<sz; i++)
    {
        switch(a[i]){
            case '(':
                push('(');
                break;
            case ')':
                pop();
                break;

            default:
                err(ERR_BAD_CHAR, "bad characters.");
        } // end of switch
    }

    return is_empty();
}
```

```
bool
paired_parentheses_switch(char a[], size_t sz) {

    make_empty();
    for (int i=0; i<sz; i++)
    {
        switch(a[i]) {
            case '(':
                push('(');
                break;
            case ')':
                pop();
                break;

            default:
                err(ERR_BAD_CHAR, "bad characters.");
        } // end of switch
    }

    return is_empty();
}
```

The diagram illustrates the execution of a `break` statement within a `switch` statement. A red curved arrow originates from the `break` statement in the `default` case of the `switch(a[i])` block and points to the closing curly brace of the `for` loop. This indicates that the `break` statement exits the `switch` and continues the execution of the loop.

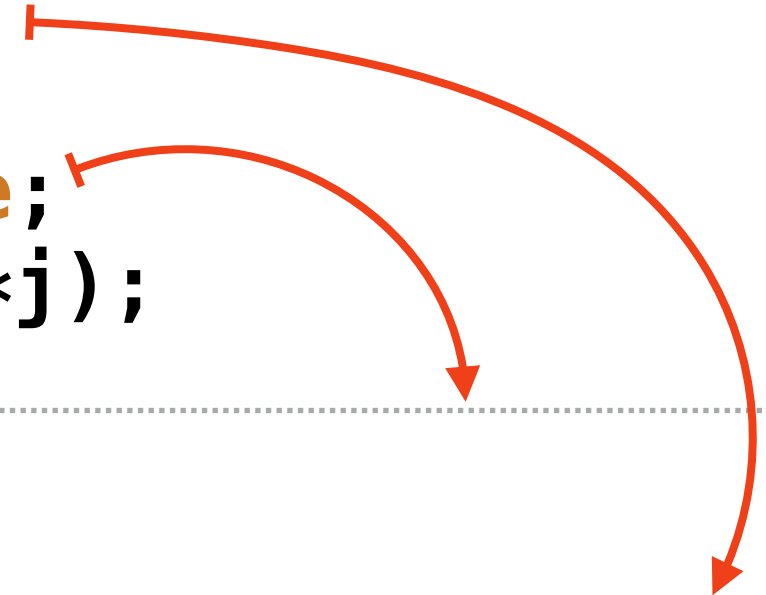
跳出switch语句

break

```
for (int i=0; i<10; i++) {  
    for (int j=0; j<10; j++) {  
        if (j==8) break;  
  
        if(j==3) continue;  
        printf("%d\n", i*j);  
    }  
}
```

continue

break



```
int main (int argc, char* argv[]) {  
    char a[] = "((()()))((((())))";  
    size_t sz = strlen(a);  
  
    assert(paired_parentheses(a, sz));  
    assert(paired_parentheses_switch(a, sz));  
  
    return 0;  
}
```

课堂练习

什么是函数的形式参数？ 什么又是函数的实际参数？

课堂练习

研读栈的实现代码，思考“队列”应该有怎样的接口？

课堂练习

括号匹配的程序中用到了一个栈，如果某个程序中，我们需要用两个栈，怎么办？

抽象数据类型

函数 与 复合数据类型

func()

unsigned double
char signed
long short

存储操作

static extern
int auto register
= unsigned int

& ~ -(单目)

运算符与表达式

+ - * / % ! || && == != < > <= >=

switch continue

程序流程控制

break while
goto if...else... for

库函数

cos
fabs

putchar

printf

strlen



WENZHENG COLLEGE OF SOOCHOW UNIVERSITY

2017.3.29



Soochow University

附录

