What C/C++ programmers need to understand about the call stack

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

March 19, 2016

Good and bad examples of stack access

Stack frames on the call stack

Call by reference and pointers into stack frames

C++ lambda expressions and stack variables

A taste of compiling

The call stack and C

- in C/C++, you need to understand how the language works
- we have seen the malloc/free on the heap, valgrind
- another part of memory is the (call) stack
- in C/C++ you can get memeory errors by misusing the stack
- (almost) all languages use a call stack
- understanding the stack is useful CS knowledge independent of C
- in compiling, stacks are central
- in OS, you have multiple call stacks
- buffer overflows target the call stack (and also heap)

scanf and &

We pass the addresses of local variables to scanf:

```
void inputadd()
{
    int x, y;
    printf("Please enter two integers:\n");
    scanf("%d", &x);
    scanf("%d", &y);
    printf("sum = %d\n", x + y);
}
```

This is fine.
But you need to be careful about pointers and the stack.

Good idea, bad idea?

```
void f()
{
    int x;
    g(&x);
}
```

Good idea, bad idea?

```
int *f()
{
    int x;
    return &x;
}
```

Good idea, bad idea?

```
int *f()
{
    int *p = malloc(sizeof(int));
    return p;
}
```

What is the scope of p?
What is the lifetime of p?
What is the lifetime of what p points to?

Good idea, bad idea?

```
void f()
{
    int x;
    int **p = malloc(sizeof(int*));
    *p = &x;
}
```

What is the scope of p?
What is the lifetime of p?
What is the lifetime of what p points to?

Good idea, bad idea?

```
void f()
{
    int x;
    free(&x);
}
```

Some terminology

- "Undefined behaviour" means that the C language gives no guarantee about what will happen. In practice, it depends on the compiler and runtime system.
- Undefined behaviour does not mean the program must crash (e.g., segmentation fault). It may crash. It may do damage.
- "Memory corruption" means that accessing (some part of) memory causes undefined behaviour.
- A pointer is called "dangling" if dereferencing it causes undefined behaviour (in the sense of the C standard). For example, taking 42 and casting it to int pointer type produces a dangling pointer.
- Undefined behaviour is the cause of many attacks, e.g., buffer overflow.
- In Java, you only get uncaught exceptions, not memory corruption.

Stack frame details

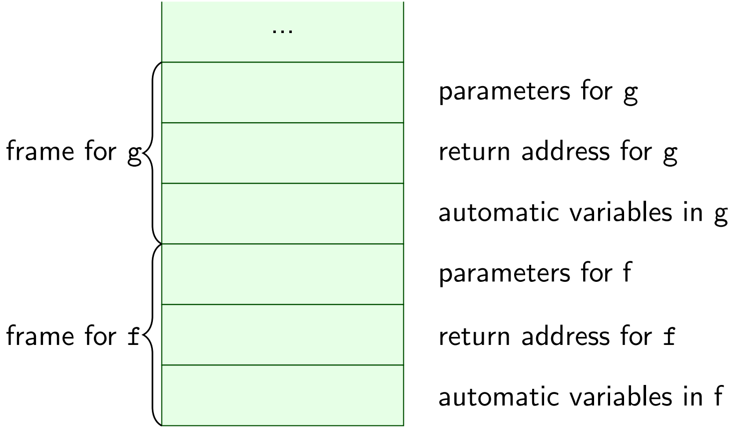
The details differ between architectures (e.g., x86, ARM, SPARC)
Ingredients of stack frames, in various order, some may be missing:
return address
parameters
local vars
saved frame pointer
caller or callee saved registers
static link (in Pascal and Algol, but not in C)
this pointer for member functions (in C++)

Naive calling convention: push args on stack

Push parameters
Then call function; this pushes the return address
This works.
It makes it very easy to have variable number of arguments, like printf in C.
But: stack is slow; registers are fast.
Compromise: use registers when possible, "spill" into stack otherwise
Optimzation (-O flags) often lead to better register usage

Call stack: used by C at run time for function calls

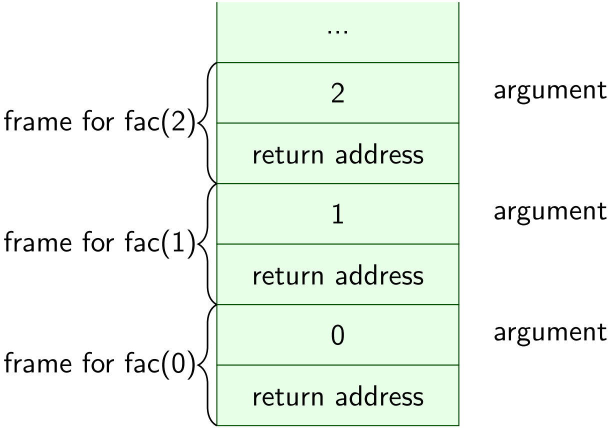
Convention: we draw the stack growing downwards on the page.
Suppose function `g` calls function `f`.



There may be more in the frame, e.g. saved registers

Call stack: one frame per function call

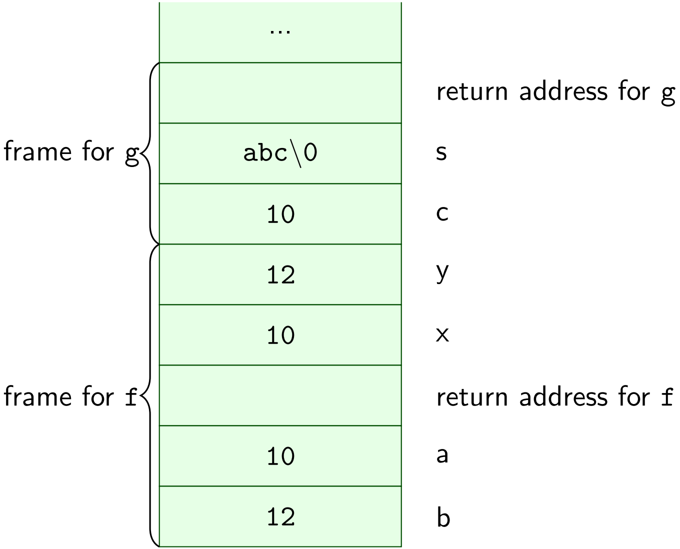
Recursion example: fac(n) calls fac(n - 1)



Call stack example code

```
int f(int x, int y) // parameters: x and y
{
    int a = x; // local variables: a and b
    int b = y;
    return a + b;
}

int g()
{
    char s[] = "abc"; // string allocated on call stack
    int c = 10;
    return f(c, c + 2);
}
```

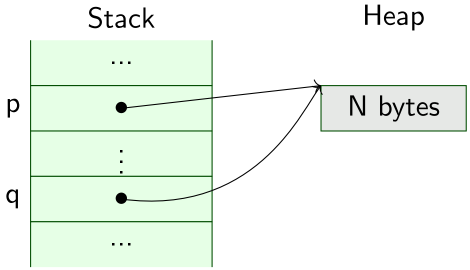
Call stack example



Call by value and pointers

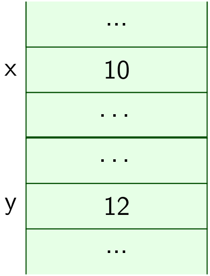
Call by value implies that a function called with a pointer gets a copy of the pointer.
What is pointed at is not copied.

```
p = malloc(N);
...
int f(char *q) { ... }
f(p)
```



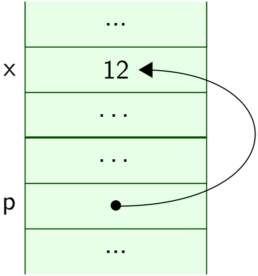
Call by value modifies only local copy

```
void f(int y)
{
    y = y + 2; // draw stack after this statement
}

void g()
{
    int x = 10;
    f(x);
}
```



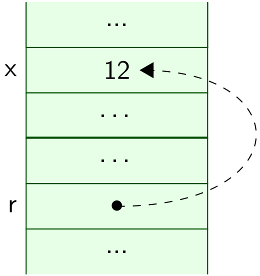
Call by reference in C = call by value + pointer

```
void f(int *p)
{
    *p = *p + 2; // draw stack after this statement
}

void g()
{
    int x = 10;
    f(&x);
}
```



Call by reference in C++

```
void f(int &r) // only C++, NOT the same as & in C
{
    r = r + 2; // draw stack after this statement
}

void g()
{
    int x = 10;
    f(x);   // the compiler passes x by reference
}
```



Pointers vs references

For a pointer p of type `int*` , we have both

```
p = q;     // change where p points
*p = 42;   // change value at the memory that p points to
```

For a reference r of type `int&`, we can only write

```
r = 42;    // change value at the memory that r points to
```

So references are less powerful and less unsafe than pointers.

Reference types in C++

It is a little confusing that the same symbol is used for the address operator in C and the reference type constructor in C++.

```
int *p = &a;        // & applied to value a in C

void f(int &r);     // & applied to type int in C++
```

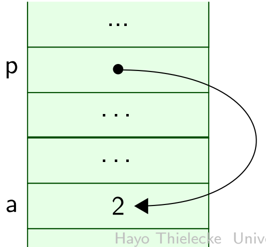
C++ is more strictly typed than C: all parameters type must be declared.

```
int main() ... // OK in C, not C++
```

One reason is that the C++compiler must know which parameters are call-by-reference
In C, all functions are call-by-value; the programmer may need to apply & when calling to pass by-reference

Returning pointer to automatic variable ☹

```
int *f()
{
    int a = 2;
    return &a; // undefined behaviour
}

void g()
{
    int *p;
    p = f();  // draw stack at this point
    printf("%d\n", *p); // may print 2, but it is undefined
}
```

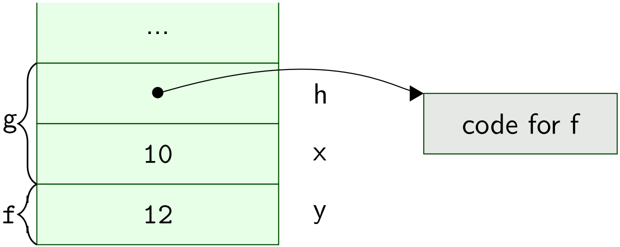


Pointers to and from stack and heap, summary

- from newer to older stack frame
  pointer passed to but not returned from function
  fine, that is how `scanf` works
- from older to newer stack frame
  pointer to auto var returned from function:
  undefined behaviour; stack frame may been reused
- from stack to heap: usually fine, unless freed to soon
- from heap to stack: usually bad, as stack frame may be reused at some point

Function pointer as function parameter

```
void g(void (*h)(int))
{
    int x = 10;
    h(x + 2);
}

void f(int y) { ... }

... g(f) ...
```



Lambdas and stack variables

```
function<int()> seta()
{
    int a = 11111 ;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

Lambdas and stack variables

```
function<int()> seta()
{
    int a = 11111 ;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

It prints 11111.

Lambdas and stack variables, by reference

```
function<int()> seta()
{
    int a = 11111 ;
    return [&] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

Lambdas and stack variables, by reference

```
function<int()> seta()
{
    int a = 11111 ;
    return [&] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

It prints 22222 when I tried it. Undefined behaviour.

Clang stack frame example

```
long f(long x, long y) // put y at -8 and x at -16
{
    long a;     // put a at -24
    long b;     // put b at -32
    ...
}
```

return addr	
old bp	← base pointer
x	← bp - 8
y	← bp - 16
a	← bp - 24
b	← bp - 32

Compiled with clang -S

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

$$
\begin{array}{lcl}
\mathtt{x} & \mapsto & \mathtt{rdi} \\
\mathtt{y} & \mapsto & \mathtt{rsi} \\
\mathtt{x} & \mapsto & \mathtt{rbp} - 8 \\
\mathtt{y} & \mapsto & \mathtt{rbp} - 16 \\
\mathtt{a} & \mapsto & \mathtt{rbp} - 24 \\
\mathtt{b} & \mapsto & \mathtt{rbp} - 32
\end{array}
$$

```
f:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq -8(%rbp), %rsi
addq $42, %rsi
movq %rsi, -24(%rbp)
movq -16(%rbp), %rsi
addq $23, %rsi
movq %rsi, -32(%rbp)
movq -24(%rbp), %rsi
imulq -32(%rbp), %rsi
movq %rsi, %rax
popq %rbp
ret
```

Optimization: compiled with clang -S -O3

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```