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

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

March 24, 2017

Some examples of stack problems

The call stack in C

Call by reference and pointers into stack frames

Good and bad examples of stack access

Function pointers on the stack

C++ lambda expressions and stack variables

A taste of compiling

Compiling structures and objects

An example from student discussion: what is the problem?

```
#include <stdlib.h>

struct s { int x; int y; };

void f(struct s *p)
{
  p = malloc(sizeof(struct s));
}

int main()
{
  struct s *p;
  f(p);
  free(p);
}
```

An example from student discussion: valgrind ☹

```
void f(struct s *p)
{
  p = malloc(sizeof(struct s));
}

int main()
{
  struct s *p;
  f(p);
  free(p);
}
```

```
==23446== LEAK SUMMARY:
==23446==    definitely lost: 8 bytes in 1 blocks
[...]
==23446== 1 errors in context 1 of 1:
==23446== Conditional jump or move depends on uninitialised
==23446==    at 0x4A063E3: free (vg_replace_malloc.c:446)
```

Fixed with more pointers ☺

```
void f(struct s **q)
{
  *q = malloc(sizeof(struct s));
}

int main()
{
  struct s *p;
  f(&p);
  free(p);
}
```

```
==23527== HEAP SUMMARY:
==23527==     in use at exit: 0 bytes in 0 blocks
==23527==   total heap usage: 1 allocs, 1 frees, 8 bytes al
==23527==
==23527== All heap blocks were freed -- no leaks are possib
==23527== ERROR SUMMARY: 0 errors from 0 contexts (suppress
```

Two bugs for the price of one ☹

The same memory management bug may have two consequences:
what should have been freed has not ⇒ memory leak
instead, something else gets freed ⇒ memory error
Valgrind reports them separately

Solution

All problems in computer science can be solved by another level of indirection.

David Wheeler

Fixed with C++ references ☺

```
struct s { int x; int y; };

void f(struct s *&q)
{
  q = new s;
}

int main()
{
  struct s *p;
  f(p);
  delete p;
}
```

Here `p` is passed by reference.

Code often seen in exams ☹

```
#include <stdlib.h>

int main()
{
  struct s x;
  // ...
  free(&x);
}
```

Code often seen in exams ☹

```
#include <stdlib.h>

int main()
{
  struct s x;
  // ...
  free(&x);
}
```

```
==23655== Invalid free() / delete / delete[] / realloc()
==23655==    at 0x4A06430: free (vg_replace_malloc.c:446)
==23655==    by 0x4004C3: main (in /home/staff/hxt/code/c/c
==23655==  Address 0x7fefffa1c is on thread 1's stack
```

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)

Stack frame details

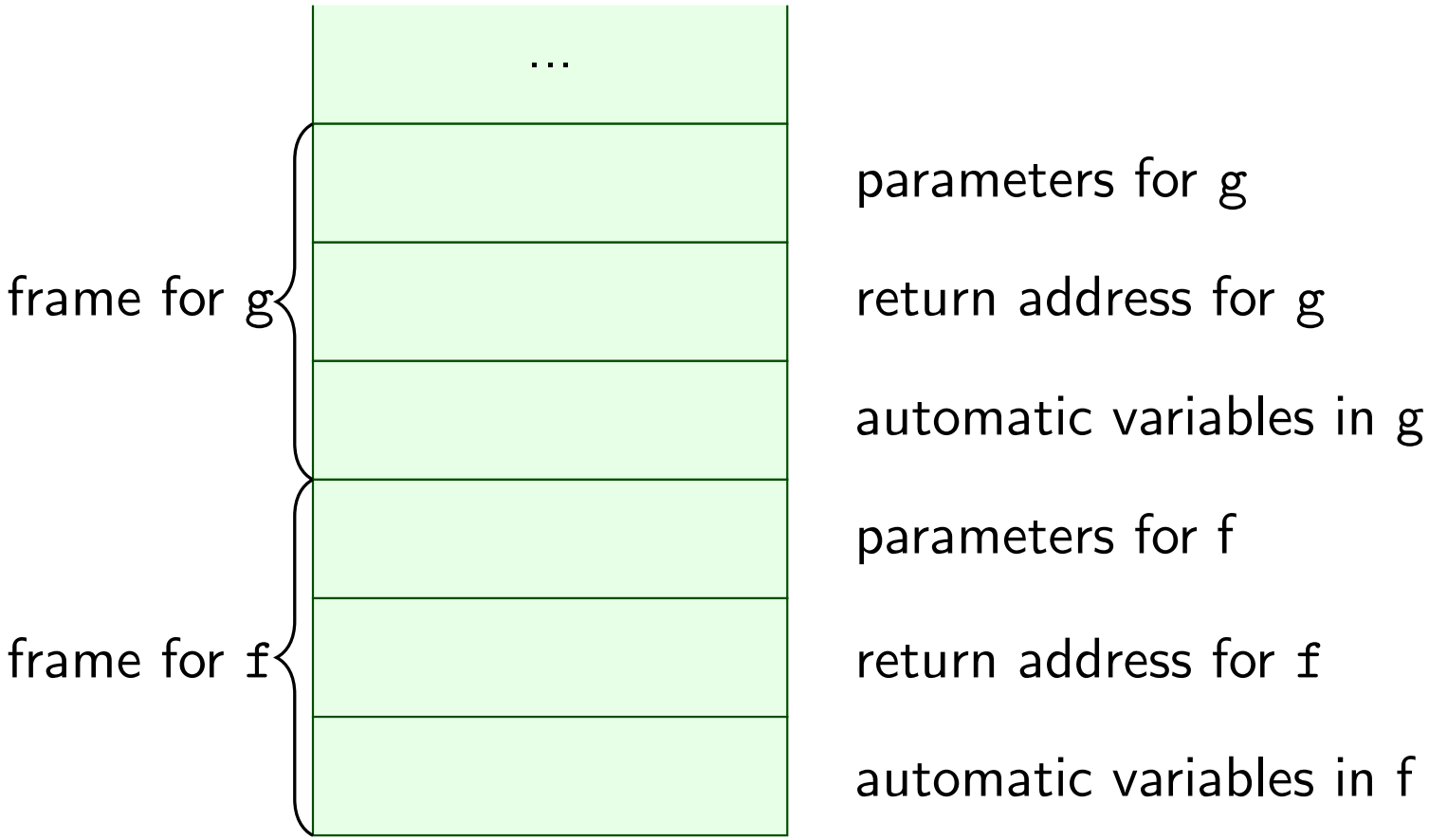
The details differ between architectures (e.g., x86, ARM, SPARC)
Ingredients of stack frames, in various order, some may be missing:
return address
parameters
local vars
saved frame pointer
caller or callee saved registers
static link (in Pascal and Algol, but not in C)
this pointer for member functions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

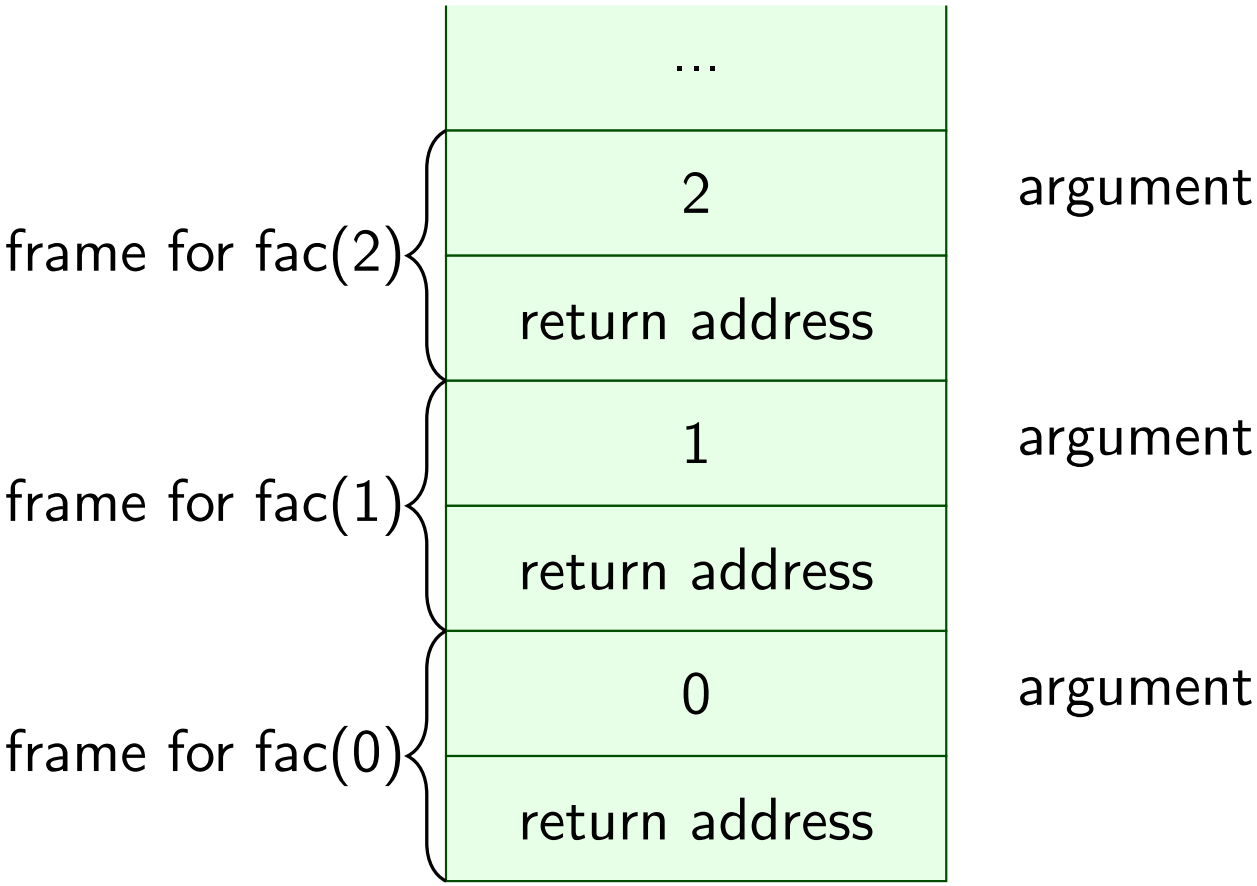
Convention: we draw the stack growing downwards on the page.
Suppose function `g` calls function `f`.



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

Call stack: one frame per function call

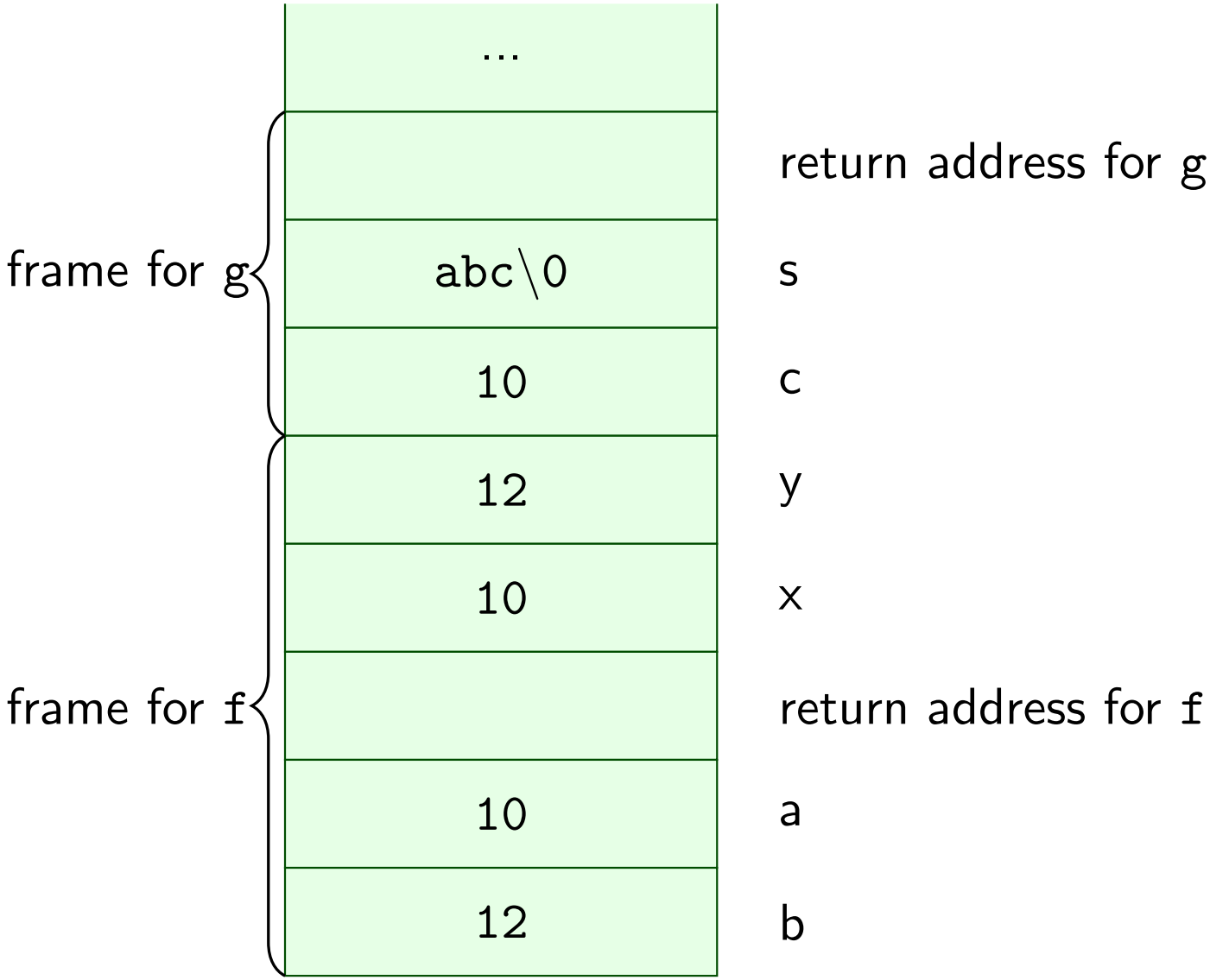
Recursion example: fac(n) calls fac(n - 1)



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);
}
```

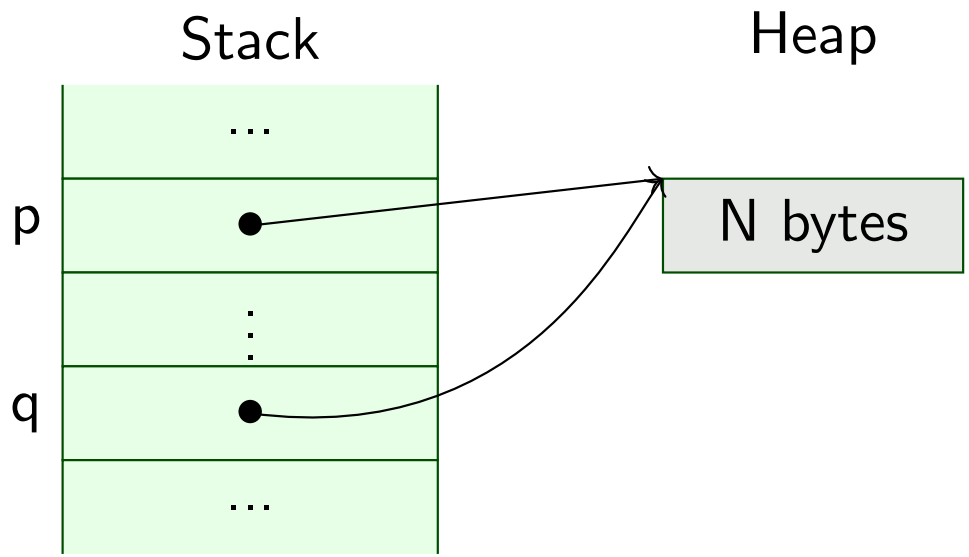
Call stack example



Call by value and pointers

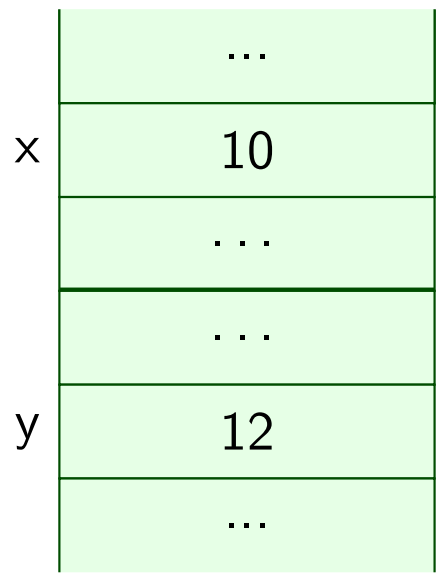
Call by value implies that a function called with a pointer gets a copy of the pointer.
What is pointed at is not copied.

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



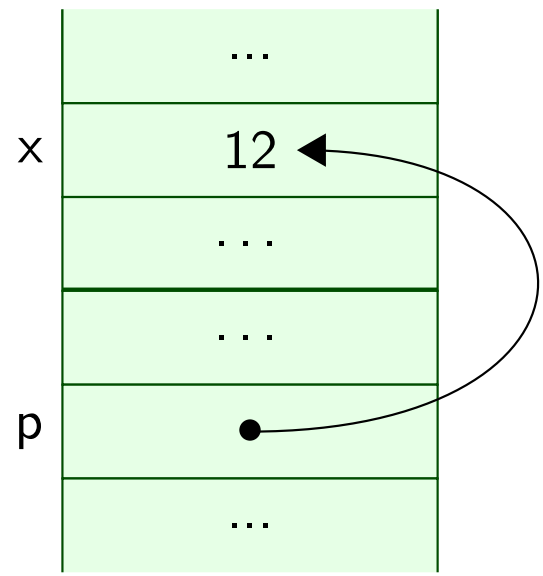
Call by value modifies only local copy

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

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



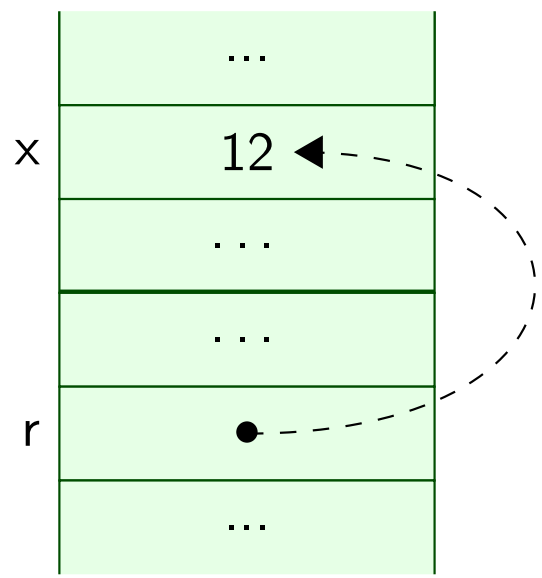
Call by reference in C = call by value + pointer

```
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++
```

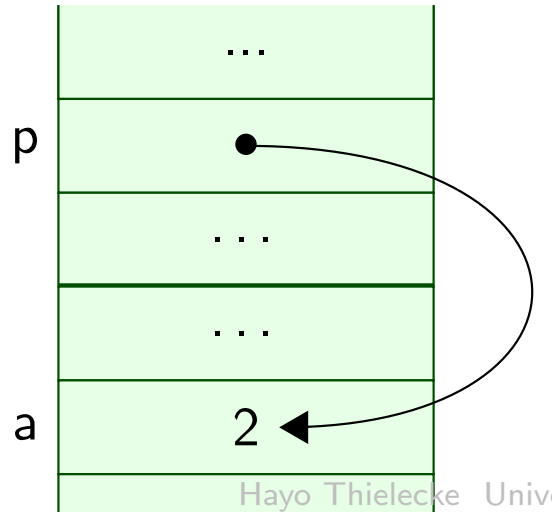
C++ is more strictly typed than C: all parameters type must be declared.

```
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

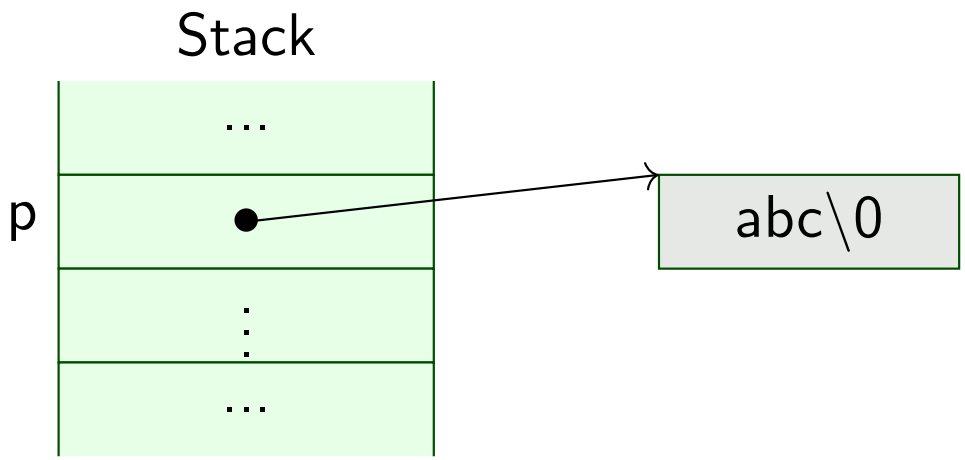
Returning pointer to automatic variable ☹

```
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
}
```



C strings are passed as pointers

```
void f(char *p) { ... };

char str[] = "abc";

f(str);
```



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 as far as the stack goes
But: no bounds checking! Do not use scanf on untrusted input

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?

```
int x;

free(&x);
```

Pointers to and from stack and heap, dos and don'ts

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:
usually bad, stack frame may been reused

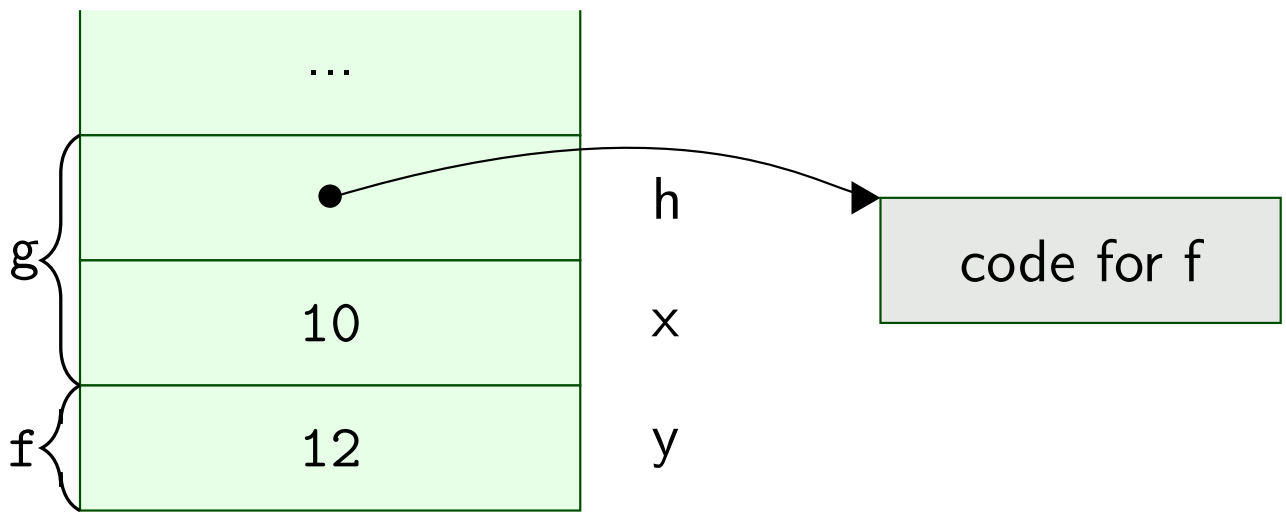
from heap to stack: usually bad, as stack frame may be reused at some point

from stack to heap: usually fine, unless freed to soon

from heap to heap: usually fine, unless freed to soon
e.g. linked list in heap

Function pointer as function parameter

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

void f(int y) { ... }

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



Non-executable stack and buffer overflow

- Function pointers can be on the call stack
- Code (compiled binaries) is not on the C call stack
- $\Rightarrow$ buffer overflow defence:
  non-executable stack, sometimes called W^X
- attacker could write binary into an array on the call stack
- attacker overwrites return address to point to injected code
- But: jumping to the injected code crashes due to non-executable permission
- classic attack from "Smashing the Stack for Fun and Profit" no longer works ☺
- "return-oriented programming" attacks use only pointers on the stack ☹

Lambdas and stack variables, capture by value ☺

```
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, capture by value ☺

```
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, capture 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, capture by reference ☹

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

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

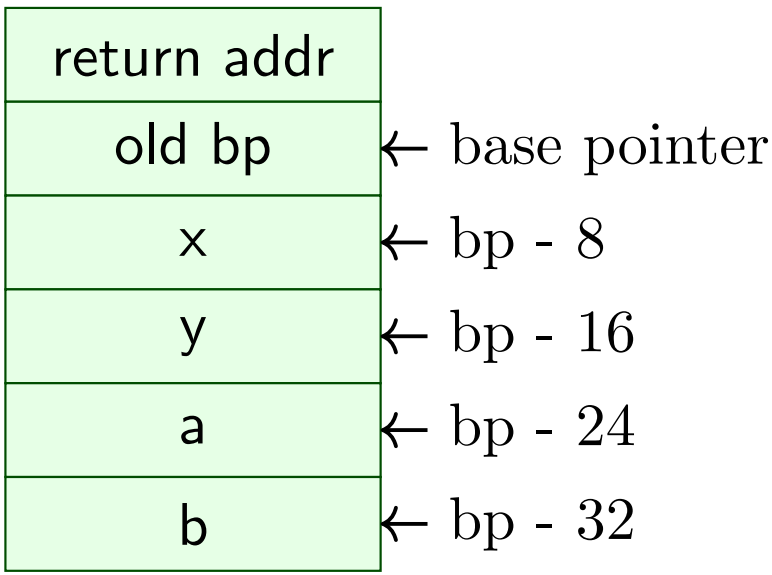
What does this print:

```
    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
    ...
}
```



Compiled with clang -S

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

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

```
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
```

Compiling structures and objects

Same idea as in stack frames:
access in memory via pointer + index
Structure definition tells the compiler the size and indices of the members.
No code is produced for a struct definition on its own.
But the compiler's symbol table is extended: it knows about the member names and their types.
Structure access then uses indexed addressing using those indices.

```
struct S {
  T1 x;
  T2 y;
};
```

Compiling structures and objects

Same idea as in stack frames:
access in memory via pointer + index
Structure definition tells the compiler the size and indices of the members.
No code is produced for a struct definition on its own.
But the compiler's symbol table is extended: it knows about the member names and their types.
Structure access then uses indexed addressing using those indices.

```
struct S {
  T1 x;
  T2 y;
};
```

`x` is a index 0
`y` is at `sizeof(T2)` + padding for alignment

Structure access

```
struct S {
  long x;
  long y;
};

void s(struct S *p)
{
  p->x = 23;
  p->y = 45;
}
```

```
s:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    $23, (%rdi)
    movq    $45, 8(%rdi)
    popq    %rbp
    retq
```

$$\begin{aligned} x &\mapsto 0 \\ y &\mapsto 8 \end{aligned}$$

Easter break yay