Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Switch-Statement Example

- Multiple case labels  
  - here: 5 & 6
- Fall-through cases  
  - here: 2
- Missing cases  
  - here: 4

long switch_eg
(long x, long y, long z) {
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
        }
    return w;
}
Supplied by CMU.

The translation is “approximate” because C doesn’t have the notion of the target of a goto being a variable. But, if it did, then the translation is what we’d want!
Supplied by CMU, but converted to x86-64.

Note that the `ja` in the slide causes a jump to occur if the previous comparison is interpreted as being performed on unsigned values, and the result is that `x` is greater than (above) 6. Given that `x` is declared to be a `signed` value, for what range of values of `x` will `ja` cause a jump to take place?

Note that the assembler code shown in the examples was produced by compiling the C code using gcc with the “-O1” flag.
Switch-Statement Example

```c
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
    . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
.align 4
.L7:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L4 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L6 # x = 5
    .quad .L6 # x = 6
```

Setup:

```
switch_eg:
    . . . # Setup
    movq %rdx, %rcx # %rcx = z
    cmpq $6, %rdi # Compare x:6
    ja .L8 # If unsigned > goto default
    jmp *.L7(%rdi,8) # Goto *JTab[x]
```

Supplied by CMU, but converted to x86-64.
The `jmp` instruction is doing a couple things that require explanation: The asterisk means it’s an indirect jump (such indirection is allowed only in jumps). The address specified after the asterisk is the address of an entry in the jump table. The asterisk means, rather than jumping directly to that entry, jump to the address that’s in that table entry. “.L7” is a label that’s being used as a displacement in the address computation. The value of .L7 is the address of the area of memory it labels. In this case, it’s the address of the jump table. Thus, an unconditional jump is to take place to the address contained in the 8-byte entry of the jump table indexed by the contents of %rdi. Thus, if %rdi is, say, 2, then a jump will take place to address in the location starting 16 bytes beyond the beginning of the table. This will be a jump to .L4. .L4 itself is a label of code specified elsewhere, the reference to the label is replaced by the assembler with the address of the code labelled with .L4.

The jump table is separate from the code (it’s not executable). This is specified by the “.section” directive, which also specifies that it should be placed in memory that’s made read-only (“.rodata” indicates this). The “.align 4” says that the address of the start of the table should be divisible by four (why this is important is something we’ll get to in a week or two).
Jump Table

Jump table

```assembly
.section .rodata
.align 4
.L7:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L4 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L6 # x = 5
.quad .L6 # x = 6
```

```c
switch(x) {
    case 1:   // .L3
        w = y*z;
        break;
    case 2:   // .L4
        w = y/z;
        /* Fall Through */
    case 3:   // .L9
        w += z;
        break;
    case 5:
    case 6:   // .L6
        w -= z;
        break;
    default:  // .L8
        w = 2;
}
```
Supplied by CMU, but converted to x86-64.
Handling Fall-Through

```c
long w = 1;
...
switch(x) {
  ...
  case 2:
    w = y/z;
    /* Fall Through */
  case 3:
    w += z;
    break;
  ...
}

  case 2:
    w = y/z;
    goto merge;

  case 3:
    w = 1;

  merge:
    w += z;
```

Supplied by CMU, but converted to x86-64.
Supplied by CMU, but converted to x86-64.

The code following the .L4 label requires some explanation. The `idivq` instruction is special in that it takes a 128-bit dividend that is implicitly assumed to reside in registers `rdx` and `rax`. Its single operand specifies the divisor. The quotient is always placed in the `rax` register, and the remainder in the `rdx` register. In our example, `y`, which we want to be the dividend, is copied into both the `rax` and `rdx` registers. The `sarq` (shift arithmetic right quadword) instruction propagates the sign bit of `rdx` across the entire register, replacing its original contents. Thus, if one considers `rdx` to contain the most-significant bits of the dividend and `rax` to contain the least-significant bits, the pair of registers now contains the 128-bit version of `y`. The `idivq` instruction computes the quotient from dividing this 128-bit value by the 64-bit value contained in register `rcx` (containing `z`). The quotient is stored register `rax` (implicitly) and the remainder is stored in register `rdx` (and is ignored in our example). This illustrated in the next slide.
The diagram illustrates the operation of the `idivq` instruction. The input `rax` is divided by the input `rcx` to produce the output `rax` and `rdx`.

- **Input**: `rax` and `rdx`
- **Output**: `rax`, `rdx`, and `rcx`
- **Dividend**: `rax` and `rdx`
- **Divisor**: `rcx`
- **Quotient**: `rax`
- **Remainder**: `rdx`
Disassembly was accomplished using “objdump –d”. Note that the text enclosed in angle brackets ("<", ">") is essentially a comment, relating the address (4004e5) to a symbolic location (0x39 bytes after the beginning of switch_eg).
Supplied by CMU, but converted to x86-64. We assume that the switch_eg function was included in a program whose name is *switch*. Hence, *gdb* is invoked from the shell with the argument “switch”.

---

**x86-64 Object Code (cont.)**

- Jump table
  - doesn’t show up in disassembled code
  - can inspect using *gdb*

```
gdb switch
(gdb) x/7xg 0x4005c0
   » examine *z* hexadecimal format “giant” words (8-bytes each)
   » use command “help x” to get format documentation
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0:</td>
<td>0x000000004004e5</td>
<td>0x000000004004bc</td>
</tr>
<tr>
<td>0x4005d0:</td>
<td>0x000000004004c4</td>
<td>0x000000004004d3</td>
</tr>
<tr>
<td>0x4005e0:</td>
<td>0x000000004004e5</td>
<td>0x000000004004dc</td>
</tr>
<tr>
<td>0x4005f0:</td>
<td>0x000000004004dc</td>
<td></td>
</tr>
</tbody>
</table>
x86-64 Object Code (cont.)

- Deciphering jump table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0</td>
<td>0x4004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4005c8</td>
<td>0x4004bc</td>
<td>1</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x4004c4</td>
<td>2</td>
</tr>
<tr>
<td>0x4005d8</td>
<td>0x4004d3</td>
<td>3</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x4004e5</td>
<td>4</td>
</tr>
<tr>
<td>0x4005e8</td>
<td>0x4004dc</td>
<td>5</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x4004dc</td>
<td>6</td>
</tr>
</tbody>
</table>

Supplied by CMU, but converted to x86-64.
### Disassembled Targets

```plaintext
(gdb) disassemble 0x4004bc,0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb
  0x000000000004004bc <switch_eg+16>: mov %rsi,%rax
  0x000000000004004bf <switch_eg+19>: imul %rdx,%rax
  0x000000000004004c3 <switch_eg+23>: retq
  0x000000000004004c4 <switch_eg+24>: mov %rsi,%rax
  0x000000000004004c7 <switch_eg+27>: mov %rsi,%rdx
  0x000000000004004ca <switch_eg+30>: sar $0x3f,%rdx
  0x000000000004004ce <switch_eg+34>: idiv %rcx
  0x000000000004004d1 <switch_eg+37>: jmp 0x4004d8 <switch_eg+44>
  0x000000000004004d3 <switch_eg+39>: mov $0x1,%eax
  0x000000000004004d8 <switch_eg+44>: add %rcx,%rax
  0x000000000004004db <switch_eg+47>: retq
  0x000000000004004dc <switch_eg+48>: mov $0x1,%eax
  0x000000000004004e1 <switch_eg+53>: sub %rdx,%rax
  0x000000000004004e4 <switch_eg+56>: retq
  0x000000000004004e5 <switch_eg+57>: mov $0x2,%eax
  0x000000000004004ea <switch_eg+62>: retq
```
Quiz 1

What C code would you compile to get the following assembler code?

```
movl $0, %eax
.L2:
  movl %eax, a(%rax,4)
  addq $1, %rax
  cmpq $10, %rax
  jne .L2
  ret
```

```
int a[10];
void func() {
  int i;
  for (i=0; i<10; i++)
    a[i]= i;
}
```

```
int a[10];
void func() {
  int i=0;
  while (i<10)
    a[i]= i++;
}
```

```
int a[10];
void func() {
  int i=0;
  switch (i) {
    case 0:
      a[i] = 0;
      break;
    default:
      a[i] = 10
  }
}
```
IA32 Stack

- Region of memory managed “last-in, first-out”
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  – address of “top” element

Stack pointer: `%esp`

Stack “bottom”

Increasing addresses

Stack grows down
IA32 Stack: Push

- `pushl src`
  - fetch operand at `src`
    - immediate, register, or memory location
  - decrement `%esp` by 4
  - store operand at address given by `%esp`

Supplied by CMU.
IA32 Stack: Pop

- `popl dest`
  - fetch operand from address given by `%esp`
  - put operand in dest
    - register or memory location
  - increment `%esp` by 4

Stack pointer: `%esp`

Supplied by CMU.
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call sub`
  - push return address on stack
  - jump to `sub`
- **Return address:**
  - address of the next instruction after call
  - example from disassembly

```
804854e:  e8 3d 06 00 00  call 8048b90 <sub>
8048553:  50    pushl %eax
```
  - return address = 0x8048553
- **Procedure return:** `ret`
  - pop address from stack
  - jump to address
Supplied by CMU.
Supplied by CMU.
For the IA32 architecture, each function’s stack frame is organized as in the slide. %ebp, sometimes called the base pointer, but more generically the frame pointer, points to a standard offset within stack frame. It’s used to refer to the arguments pushed into the caller’s stack frame as well as to local variables, etc., pushed into the function’s stack frame.
The convention for the IA32 architecture is for the caller of a function to push its arguments on the stack in reverse order. It then calls the function, which has the effect of pushing the return address (the address of the instruction following the call) onto the stack.
Again, following the IA32 convention, the first thing a function does is to push the contents of %ebp onto the stack, thus saving the pointer to the caller’s stack frame. It then copies the current stack pointer (%esp) into %ebp, so that %ebp now refers to the current stack frame. Having done this, the function can now refer to its arguments via offsets from %ebp.

When the function is ready to return to its caller, it first pops off the stack the copy of the caller’s %ebp that was pushed onto the stack, replacing the current contents of %ebp with this saved value. This has the effect of making the caller’s stack frame the current frame. Next the function calls ret, which pops the return address off the stack and sets %eip (the instruction pointer) to that value, causing control to return to the caller at the instruction following the call instruction.
If the function has local variables, these are allocated on the stack by decrementing the stack pointer to account for the space needed, and then popped off the stack when the function returns by adding the space occupied back to the stack pointer.
The *leave* instruction causes the contents of ebp to be copied into esp, thereby removing everything from the stack that had been pushed into the frame. It then pops the current stack top (the old ebp) into the ebp register. The effect of *leave* is thus to return to the caller’s stack frame.

There is an *enter* instruction that has the same effect as that of the first three instructions of subr combined (it has an operand that indicates how much space for local variables to allocate). However, it’s not used by gcc, apparently because it’s slower than doing it as shown in the slide.
Register-Saving Conventions

- **When procedure you calls who:**
  - you is the **caller**
  - who is the **callee**

- **Can registers be used for temporary storage?**

  ```
  you:
  ...  
  movl $33, %edx
  call who
  addl %edx, %eax
  ...  
  ret

  who:
  ...  
  movl 8(%ebp), %edx
  addl $32, %edx
  ...  
  ret
  ```

  - contents of register %edx overwritten by who
  - this could be trouble: something should be done!

  » need some coordination
Register-Saving Conventions

- When procedure \( y \) calls \( x \):
  - \( y \) is the caller
  - \( x \) is the callee

- Can registers be used for temporary storage?

- Conventions
  - "caller save"
    - caller saves registers containing temporary values on stack before the call
    - restores them after call
  - "callee save"
    - callee saves registers on stack before using
    - restores them before returning
IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`
  - caller saves prior to call if values are used later

- `%eax`
  - also used to return integer value

- `%ebx`, `%esi`, `%edi`
  - callee saves if wants to use them

- `%esp`, `%ebp`
  - special form of callee-save
  - restored to original values upon exit from procedure

Supplied by CMU.
Register-Saving Example

**yoo:**

```
... 
movl $33, %edx
pushl %edx
call who
popl %edx
addl %edx, %eax
... 
ret
```

**who:**

```
... 
pushl %ebx
...
movl 4(%ebp), %ebx
addl %53, %ebx
movl 8(%ebp), %edx
addl $32, %edx
... 
popl %ebx
... 
ret
```
Quiz 2

- The `leave` instruction copies the current value of `%ebp` into `%esp` and then pops the stack (the old `%ebp`) into `%ebp`. It's followed by a `ret` instruction. Does this approach for returning from a procedure work if there are saved registers in the stack frame?
  
  a) always
  
  b) usually
  
  c) hardly ever
Recursive Function

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- Registers
  - `%eax`, `%edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

```assembly
pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl %eax, %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
addl $1, %edx
lea (%edx,%eax), %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
```
Recursive Call #1

/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

- Actions
  - save old value of %ebx on stack
  - allocate space for argument to recursive call
  - store x in %ebx

Supplied by CMU.
Recursive Call #2

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- if `x == 0`, return
  - with `%eax` set to 0

```asm
    ...  
movl  $0, %eax  
testl %ebx, %ebx  
je    .L3  
    ...  
.L3:    
    ...  
    ret
```

Supplied by CMU.
Recursive Call #3

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return 
        (x & 1) + pcount_r(x >> 1);
}
```

- **Actions**
  - store x >> 1 on stack
  - make recursive call

- **Effect**
  - %eax set to function result
  - %ebx still has value of x

```assembly
... movl %ebx, %eax
shr $1, %eax
movl %eax, (%esp)
call pcount_r
...
```

Supplied by CMU.
Recursive Call #4

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- **Assume**
  - %eax holds value from recursive call
  - %ebx holds x

- **Actions**
  - compute (x & 1) + computed value

- **Effect**
  - %eax set to function result

```assembly
  ..
  movl %ebx, %edx
  andl $1, %edx
  leal (%edx,%eax), %eax
  ..
```

Supplied by CMU.
Recursive Call #5

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

L3:
```
addl $4, %esp
popl %ebx
popl %ebp
ret
```

- **Actions**
  - restore values of %ebx and %ebp
  - restore %esp

Supplied by CMU.
Observations About Recursion

- **Handled without special consideration**
  - stack frames mean that each function call has private storage
    - saved registers & local variables
    - saved return pointer
  - register-saving conventions prevent one function call from corrupting another’s data
  - stack discipline follows call / return pattern
    - if P calls Q, then Q returns before P
    - last-in, first-out

- **Also works for mutual recursion**
  - P calls Q; Q calls P

Supplied by CMU.
Note that "frame pointer" is synonymous with "base pointer".

Why Bother with a Frame Pointer?

- It points to the beginning of the stack frame
  - making it easy for people to figure out where things are in the frame
  - but people don’t execute the code ...
- The stack pointer always points somewhere within the stack frame
  - it moves about, but the compiler knows where it is pointing
    » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    » tough for people, but easy for the compiler
- Thus the frame pointer is superfluous
  - it can be used as a general-purpose register
### x86-64 General-Purpose Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Supplied by CMU.

Note that the `leave` instruction is no longer relevant, since `%rbp` does not contain the address of the stack frame.
**x86-64 Long Swap**

```c
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **Operands passed in registers**
  - first (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- **No stack operations required (except ret)**
- **Avoiding stack**
  - can hold all local information in registers

---

Supplied by CMU.

Note that *swap_l* is a *leaf* function, meaning that it does not call other functions.
The `volatile` keyword tells the compiler that it may not perform optimizations on the associated variable such as storing it strictly in registers and not in memory. It's used primarily in cases where the variable might be modified via other routines that aren’t apparent when the current code is being compiled. We’ll see useful examples of its use later. Here it’s used simply to ensure that `loc` is allocated on the stack, thus giving us a simple example of using local variables stored on the stack.

The issue here is whether a reference to memory beyond the current stack (as delineated by the stack pointer) is a legal reference. On IA32 it is not, but on x86-64 it is, as long as the reference is not more than 128 bytes beyond the end of the stack.
The `movslq` instruction copies a long into a quad, propagating the sign bit into the upper 32 bits of the quad word. For example, suppose `%esi` contains 0x08888888. After the execution of `movslq %esi, %rsi, %rsi` will contain 0x0000000088888888. But if `%esi` initially contains 0x88888888 (i.e., the sign bit is set), then after execution of the instruction, `%rsi` will contain 0xffffffff88888888.
Supplied by CMU.

Note that sum is a global variable. While its exact location in memory is not known by the compiler, it will be stored in memory at some location just beyond the end of the executable code (which is known as “text”). Thus the compiler can refer to sum via the instruction pointer. The actual displacement, i.e., the distance from the current target of the instruction pointer and the location of sum, is not known to the compiler, but will be known to the linker, which will fill this displacement in when the program is linked. This will all be explained in detail in a few weeks.
Understanding x86-64 Stack Frame

```assembly
swap_ele_su:
  movq  %rbx, -16(%rsp)       # Save %rbx
  movq  %rbp, -8(%rsp)        # Save %rbp
  subq  $16, %rsp             # Allocate stack frame
  movslq %esi, %rax           # Extend i into quad word
  leaq  8(%rdi,%rax,8), %rbx  # a[i+1] (callee save)
  leaq  (%rdi,%rax,8), %rbp   # a[i]   (callee save)
  movq  %rbx, %rsi            # 2nd argument
  movq  %rbp, %rdi            # 1st argument
  call  swap
  movq  (%rbx), %rax          # Get a[i+1]
  imulq (%rbp), %rax          # Multiply by a[i]
  addq  %rax, sum(%rip)       # Add to sum
  movq  (%rsp), %rbx          # Restore %rbx
  movq  8(%rsp), %rbp         # Restore %rbp
  addq  $16, %rsp             # Deallocate frame
  ret
```

Supplied by CMU.
Understanding x86-64 Stack Frame

```
movq  %rbx, -16(%rsp)         # Save %rbx
movq  %rbp, -8(%rsp)          # Save %rbp

subq $16, %rsp                # Allocate stack frame

movq (%rsp), %rbx            # Restore %rbx
movq 8(%rsp), %rbp           # Restore %rbp
addq $16, %rsp               # Deallocate frame
```
Quiz 2

swap_e1e_s1:

movq %rbx, -16(%rsp)
movq %rbp, -8(%rsp)
subq $16, %rsp
movslq %esi,%rax
leaq 8(%rdi,%rax,8), %rbx
leaq (%rdi,%rax,8), %rbp
movq %rbx, %rsi
movq %rbp, %rdi
call swap
movq (%rbx), %rax
imulq (%rbp), %rax
addq %rax, sum(%rip)
movq (%rsp), %rbx
movq 8(%rsp), %rbp
addq $16, %rsp
ret

Since a 128-byte red zone is allowed, is it necessary to allocate the stack frame by subtracting 16 from %rsp?

a) yes
b) no
The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion*. 

```c
int factorial(int x) {
    if (x == 1)
        return x;
    else
        return x * factorial(x - 1);
}
```

```c
int factorial(int x) {
    return f2(x, 1);
}
```

```c
int f2(int a1, int a2) {
    if (a1 == 1)
        return a2;
    else
        return f2(a1 - 1, a1 * a2);
}
```
Here we look at the stack usage for the version without tail recursion. Note that we have as many stack frames as the value of the argument; the results of the calls are combined after the stack reaches its maximum size.
No Tail Recursion (2)

<table>
<thead>
<tr>
<th>x</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>720</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
With tail recursion, since the result of the recursive call is not used by the issuing stack frame, it’s possible to reuse the issuing stack frame to handle the recursive invocation. Thus rather than push a new stack frame on the stack, the current one is written over. Thus the entire sequence of recursive calls can be handled within a single stack frame.
This is the result of compiling the tail-recursive version of factorial using gcc with the \textit{–O1} flag. This flag turns on a moderate level of code optimization, but not enough to cause the stack frame to be reused.
Here we’ve compiled the program using the \texttt{-O2} flag, which turns on additional optimization (at the cost of increased compile time), with the result that the recursive calls are optimized away — they are replaced with a loop.

Why not always compile with \texttt{-O2}? For “production code” that is bug-free (assuming this is possible), this is a good idea. But this and other aggressive optimizations make it difficult to relate the runtime code with the source code. Thus, a runtime error might occur at some point in the program’s execution, but it is impossible to determine exactly which line of the source code was in play when the error occurred.