CSE351: Section 5

Procedures, Stacks,
Structs and Unions

October 27, 2011

Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers

%ebx used, but saved at beginning & restored at end

%eax used without first saving

- expect caller to save
- pushed onto stack as parameter for next call
- used for return value

Convention dictates behavior

Passing by Reference with Pointers

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

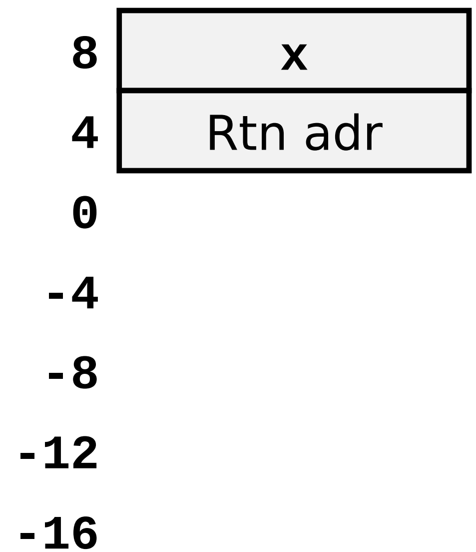
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Passing a pointer to a function allows the function to modify the contents of the memory being pointed to

Creating & Initializing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- Variable `val` must be stored on stack
- Need to pass a pointer to `s_helper`
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument



Initial part of `sfact`

```
_sfact:
   pushl %ebp
   movl %esp,%ebp
   subl $16,%esp
   movl 8(%ebp),%edx
   movl $1,-4(%ebp)
```

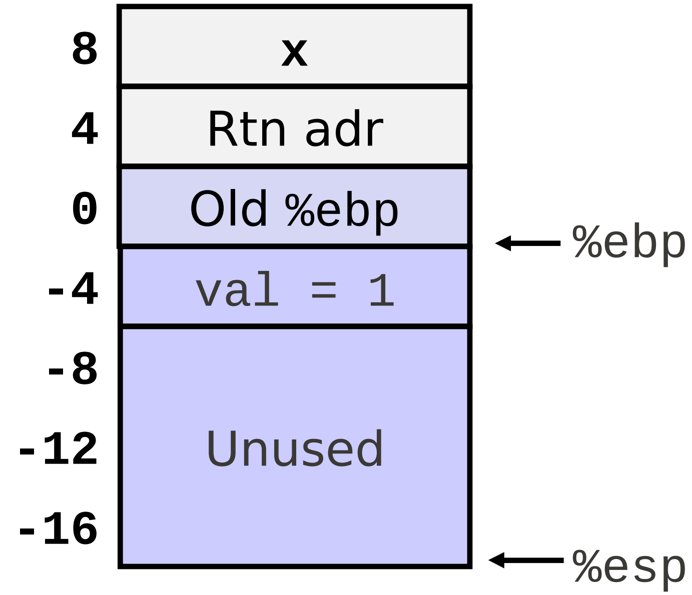
Creating & Initializing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- Variable `val` must be stored on stack
- Need to pass a pointer to `s_helper`
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument

Initial part of `sfact`

```
_sfact:
   pushl %ebp
   movl %esp,%ebp
   subl $16,%esp
   movl 8(%ebp),%edx
   movl $1,-4(%ebp)
```



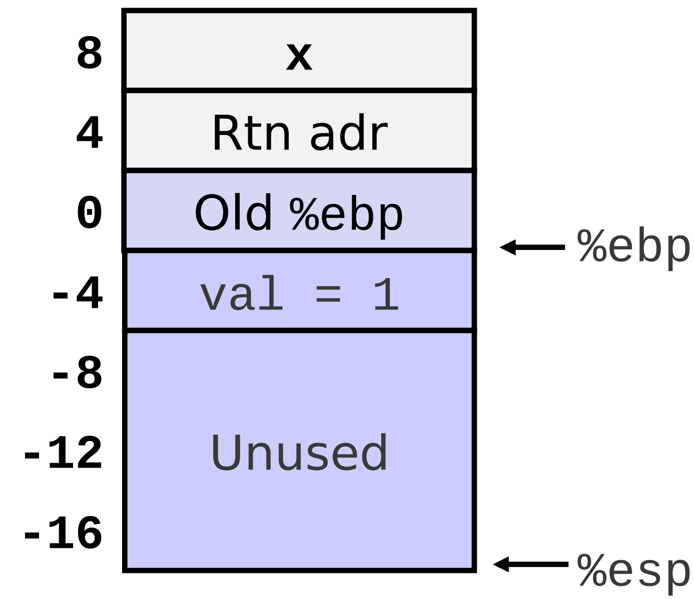
Creating & Initializing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- Variable `val` must be stored on stack
- Need to pass a pointer to `s_helper`
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument

Initial part of `sfact`

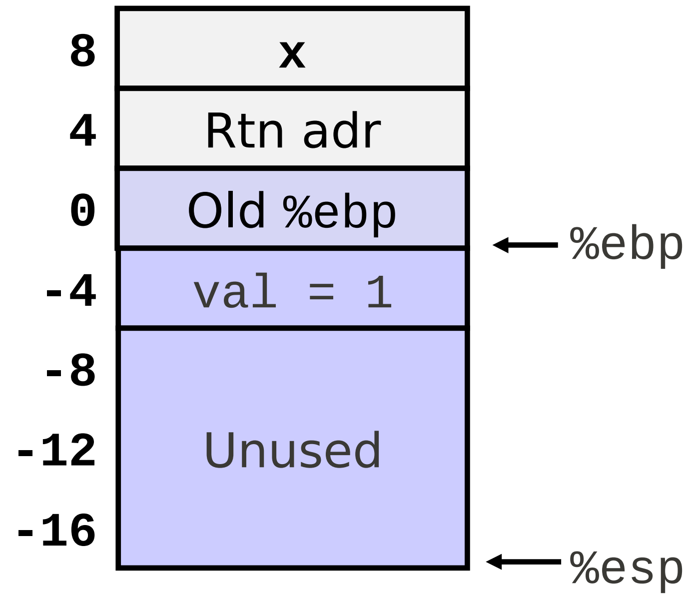
```
_sfact:
   pushl %ebp            # Save %ebp
   movl %esp,%ebp        # Set %ebp
   subl $16,%esp         # Add 16 bytes
   movl 8(%ebp),%edx     # edx = x
   movl $1,-4(%ebp)      # val = 1
```



Passing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call

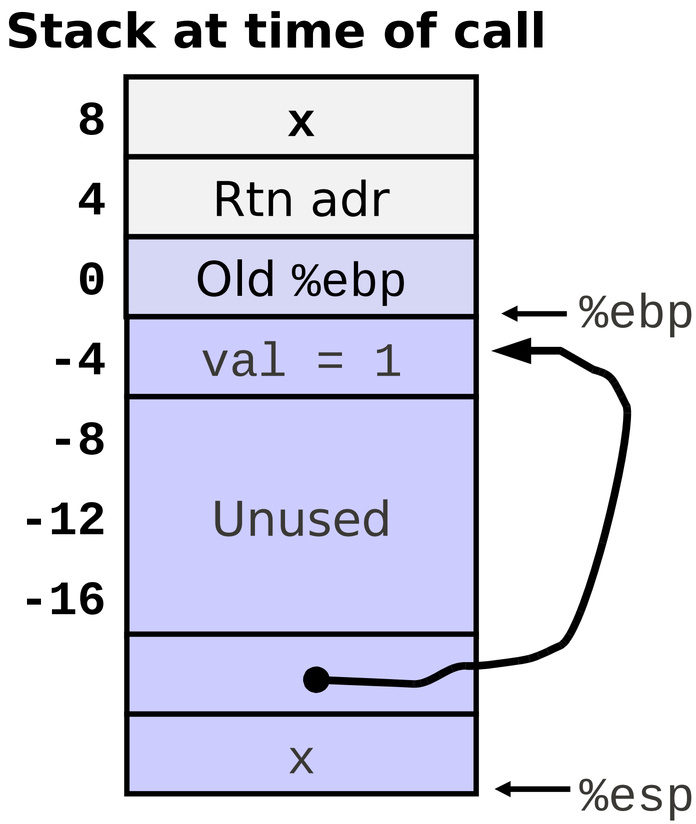


Calling `s_helper` from `sfact`

```
    leal -4(%ebp),%eax
    pushl %eax
    pushl %edx
    call s_helper
    movl -4(%ebp),%eax
    • • •
```

Passing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```



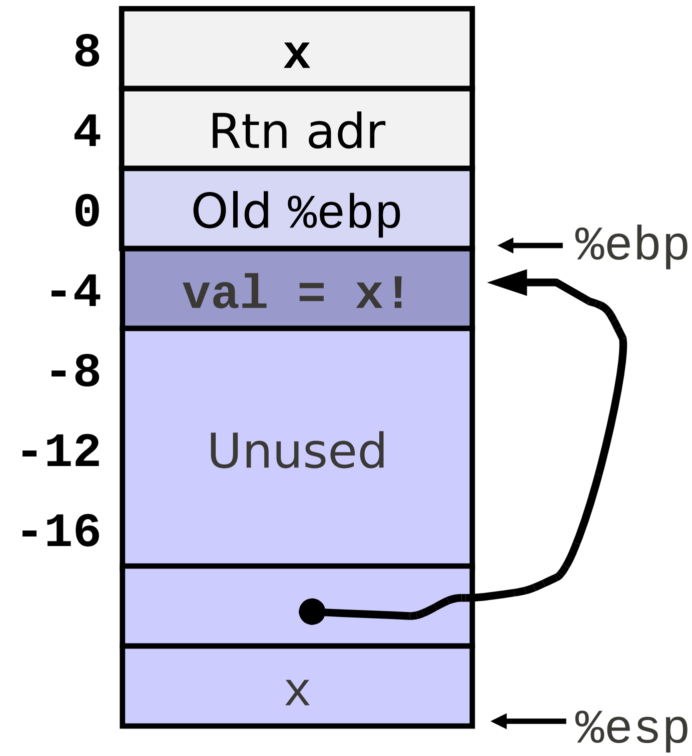
Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
• • •
```

Passing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



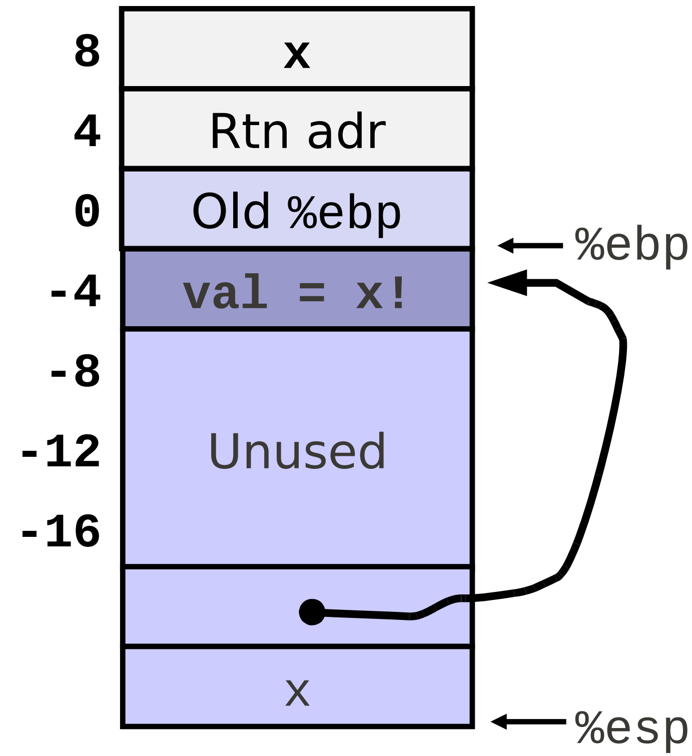
Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
• • •
```

Passing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call

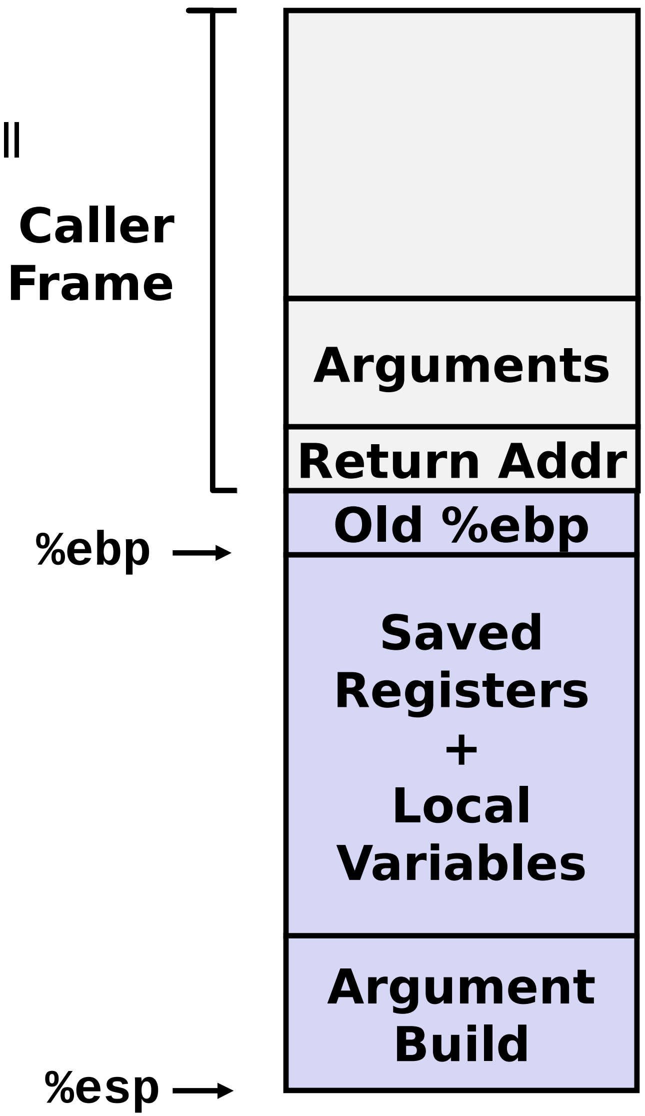


Calling s_helper from sfact

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
call s_helper       # call
movl -4(%ebp),%eax  # Return val
• • •               # Finish
```

IA 32 Procedure Summary

- Stack makes recursion work
- Private storage for each *instance* of procedure call
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Managed by stack discipline
  - Procedures return in inverse order of calls
- IA32 procedures
  - *Combination of Instructions + Conventions*
  - `call / ret` instructions
  - Register usage conventions
    - caller / callee save
    - **%ebp** and **%esp**
  - Stack frame organization conventions



x86-64 Registers: Conventions

`%rax`	Return value
`%rbx`	Callee saved
`%rcx`	Argument #4
`%rdx`	Argument #3
`%rsi`	Argument #2
`%rdi`	Argument #1
`%rsp`	Stack pointer
`%rbp`	Callee saved

`%r8`	Argument #5
`%r9`	Argument #6
`%r10`	Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

Some Differences between x8664 and IA32

- More general purpose registers
- First six function arguments passed via registers
  - Why?
- Sometimes we don't need a frame pointer
  - Why?
  - How are local variables accessed?
- Misc. differences in instructions, too

Using Nested Arrays

Strengths

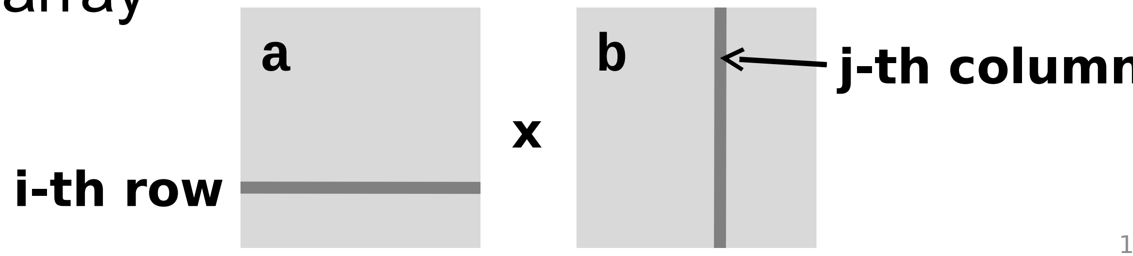
- C compiler handles doubly subscripted arrays
- Generates very efficient code
- Avoids multiply in index computation

Limitation

- Only works for fixed array size

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```



Dynamic Nested Arrays

<u>Strength</u>

- Can create matrix of any size

<u>Programming</u>

- Must do index computation explicitly

<u>Performance</u>

- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
  (int *a, int i, int j, int n)
{
  return a[i*n+j];
}
```

Dynamic Nested Arrays

Strength

- Can create matrix of any size

Programming

- Must do index computation explicitly

Performance

```
movl 12(%ebp),%eax         # i
movl 8(%ebp),%edx          # a
imull 20(%ebp),%eax        # n*i
addl 16(%ebp),%eax         # n*i+j
movl (%edx,%eax,4),%eax    # Mem[a+4*(i*n+j)]
```

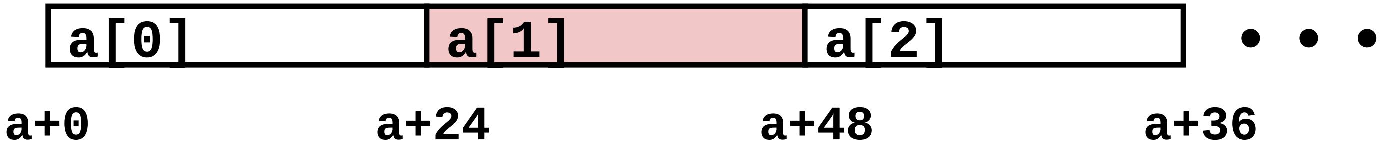
```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
  (int *a, int i, int j, int n)
{
  return a[i*n+j];
}
```

Arrays of Structures

Each element in the array must be properly aligned.

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

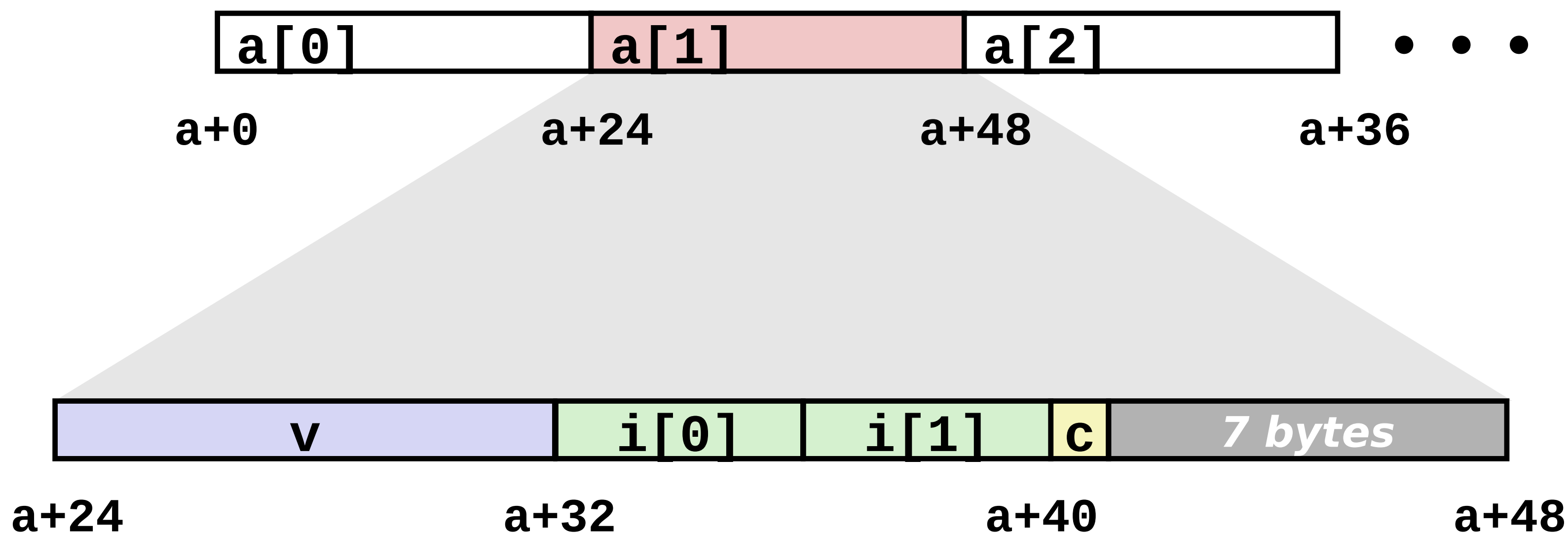


Arrays of Structures

Each element in the array must be properly aligned.

True data length is 8 + 2*4 + 1, but actually uses 8 + 2*4 + 8

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

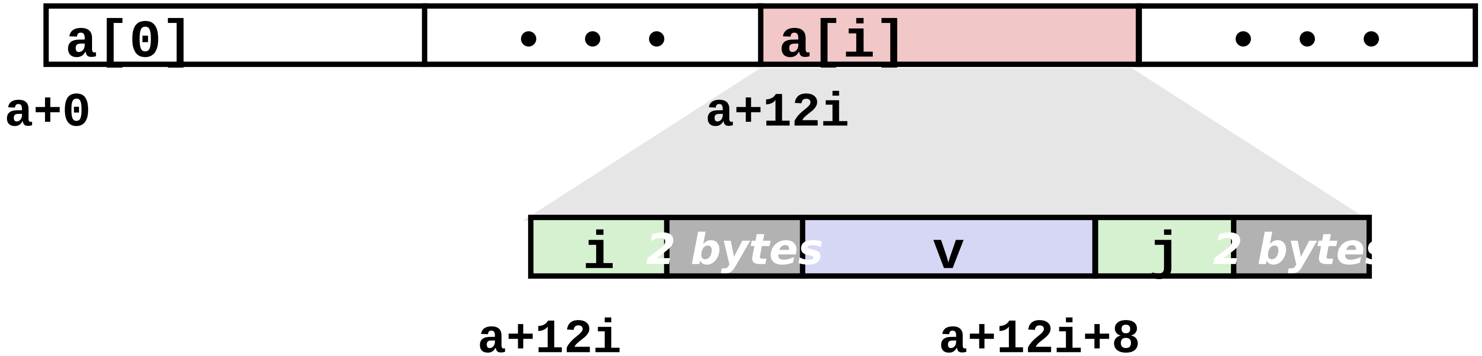


Accessing Array Elements

Struct S3 is 8 bytes, but requires 12 bytes for padding

To access the ith element in a, we compute the offset as 12*i

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx)
{
   return a[idx].j;
// return (a + idx)->j;
}
```

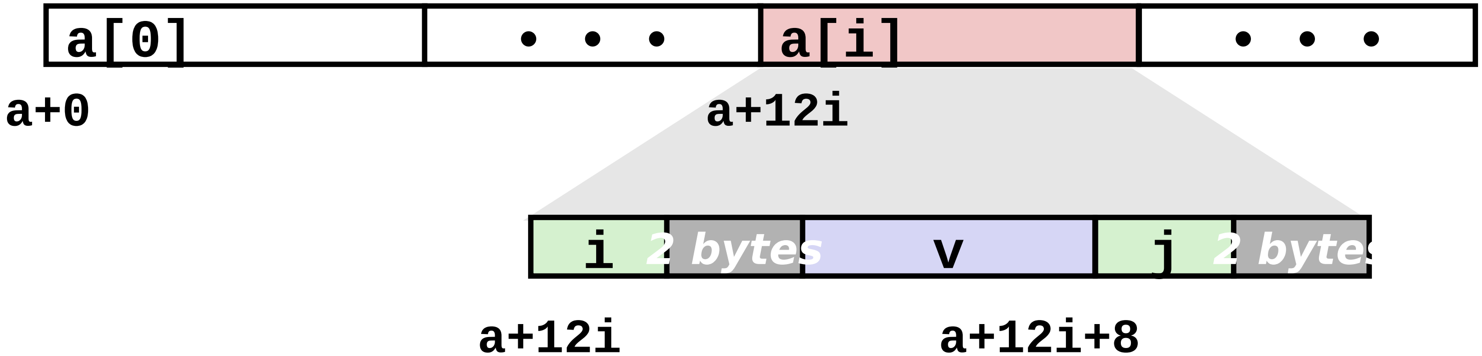
```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Accessing Array Elements

To get to member j:

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx)
{
   return a[idx].j;
// return (a + idx)->j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Unions

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

Concept

- Allow same regions of memory to be referenced as different types
- Aliases for the same memory location

Unions

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

Structure Layout

Concept

- Allow same regions of memory to be referenced as different types
- Aliases for the same memory location

Unions

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

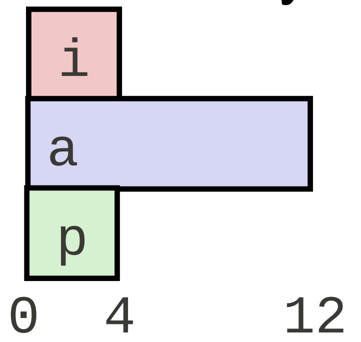
```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

Structure Layout

Concept

- Allow same regions of memory to be referenced as different types
- Aliases for the same memory location

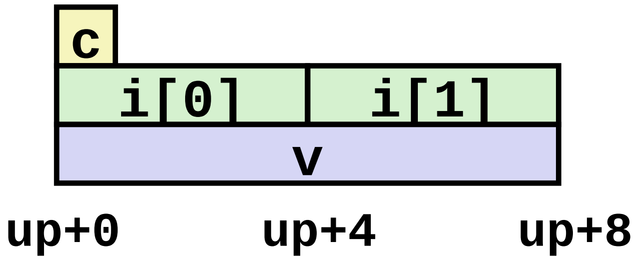
Union Layout



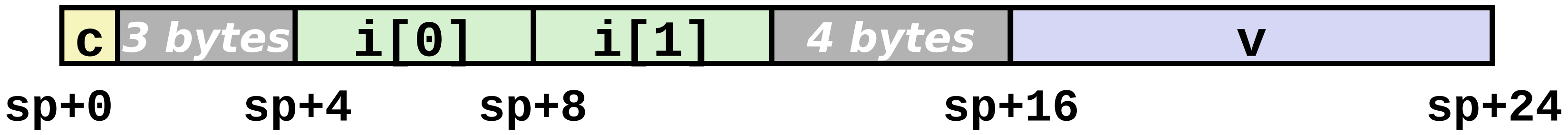
Union Allocation

- Size determined by the largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

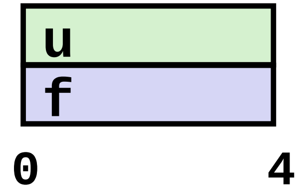



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Unions to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

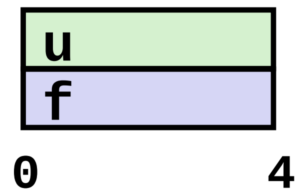
```
unsigned float2bit(float f) {
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float)u`?

Same as `(unsigned)f`?

Using Unions to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f) {
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float)u`?

Same as `(unsigned)f`?

No! Casts actually trigger a bit conversion

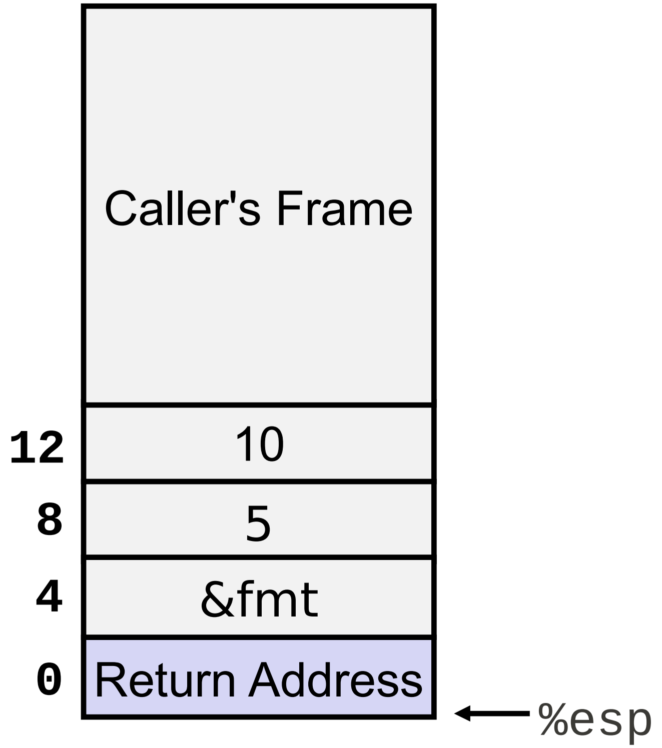
Variable-Argument Functions

- How many arguments does `printf` take?
  - As many as we want!
- What does the function signature look like?
  - `int printf(const char *fmt, …)`
  - The "`...`" tells compiler to expect a variable number of arguments
- How do we pass an arbitrary number of arguments?
  - Just push 'em all on the stack like before
- How does `printf` know how many arguments it received?
  - The format string tells it what to expect and in what order

Variable-Argument Functions

- Example: `printf("%d %d\n", 5, 10)`

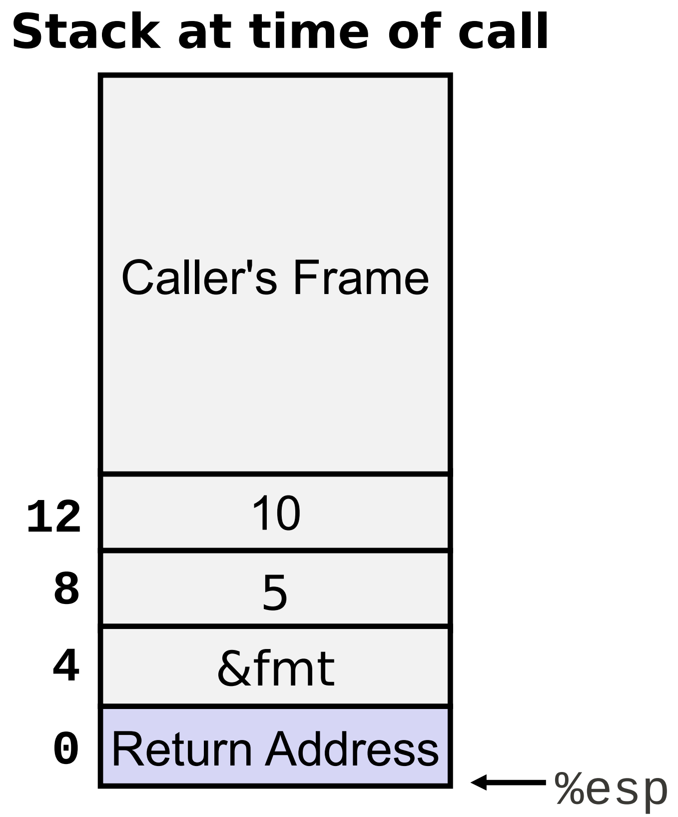
Stack at time of call



Output: `"5 10"`

Variable-Argument Functions

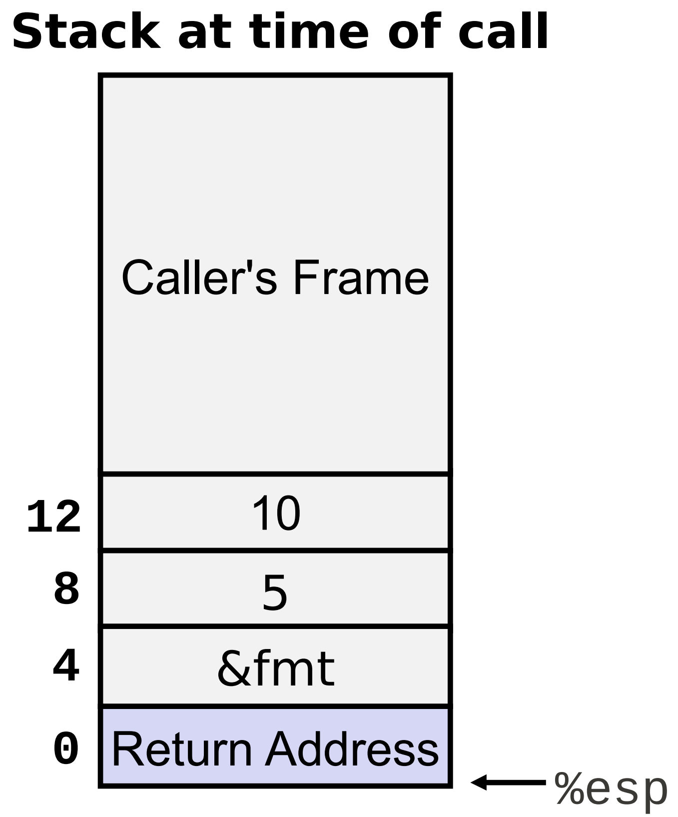
- Example: `printf("%d %d %d %d\n", 5, 10)`



Output?

Variable-Argument Functions

- Example: `printf("%d %d %d %d\n", 5, 10)`



Output: `"5 10 ??????? ???????"`