15-213
“The course that gives CMU its Zip!”
Structured Data:
Sept. 20, 2001

Topics
• Arrays
• Structs
• Unions
Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Type</th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td></td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td></td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td></td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td></td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td></td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td></td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array Allocation

Basic Principle

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L * \text{sizeof}(T) \) bytes

```
char string[12];
int val[5];
double a[4];
char *p[3];
```
Array Access

Basic Principle

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Identifier \( A \) can be used as a pointer to starting element of the array

\[
\begin{array}{c}
\text{Reference} & \text{Type} & \text{Value} \\
\text{val[4]} & \text{int} & 3 \\
\text{val} & \text{int *} & x \\
\text{val+1} & \text{int *} & x + 4 \\
\text{&val[2]} & \text{int *} & x + 8 \\
\text{val[5]} & \text{int} & ?? \\
*(\text{val+1}) & \text{int} & 5 \\
\text{val + i} & \text{int *} & x + 4 \ i \\
\end{array}
\]
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

Notes

- Declaration "zip_dig cmu" equivalent to "int cmu[5]"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

Computation

- Register $%edx$ contains starting address of array
- Register $%eax$ contains array index
- Desired digit at $4 \times %eax + %edx$
- Use memory reference $(%edx, %eax, 4)$

Memory Reference Code

```c
int get_digit (zip_digit z, int dig)
{
    return z[dig];
}
```

```c
# %edx = z
# %eax = dig
movl (%edx, %eax, 4), %eax # z[dig]
```
## Referencing Examples

### Code Does Not Do Any Bounds Checking!

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mit[3]</code></td>
<td><code>36 + 4* 3 = 48</code></td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td><code>mit[5]</code></td>
<td><code>36 + 4* 5 = 56</code></td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td><code>mit[-1]</code></td>
<td><code>36 + 4* -1 = 32</code></td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td><code>cmu[15]</code></td>
<td><code>16 + 4*15 = 76</code></td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- **Out of range behavior implementation-dependent**
  - No guaranteed relative allocation of different arrays
Array Loop Example

Original Source

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

Transformed Version

- As generated by GCC
- Eliminate loop variable i
- Convert array code to pointer code
- Express in do-while form
  - No need to test at entrance
Array Loop Implementation

Registers
- %ecx  z
- %eax  zi
- %ebx  zend

Computations
- \(10 \times zi + *z\) implemented as \(*z + 2 \times (zi + 4 \times zi)\)
- \(z++\) increments by 4

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax  # zi = 0
leal 16(%ecx),%ebx  # zend = z+4
.L59:
    leal (%eax,%eax,4),%edx  # 5*zi
movl (%ecx),%eax  # *z
addl $4,%ecx  # z++
    leal (%eax,%edx,2),%eax  # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx  # z : zend
jle .L59  # if <= goto loop
```
Nested Array Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3},
     {1, 5, 2, 1, 7},
     {1, 5, 2, 2, 1}};
```

- Declaration “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
  - Variable `pgh` denotes array of 4 elements
    » Allocated contiguously
  - Each element is an array of 5 int’s
    » Allocated contiguously
- “Row-Major” ordering of all elements guaranteed
Nested Array Allocation

Declaration

\[ T \ A[R][C]; \]
- Array of data type \( T \)
- \( R \) rows
- \( C \) columns
- Type \( T \) element requires \( K \) bytes

Array Size

- \( R \times C \times K \) bytes

Arrangement

- Row-Major Ordering

\[
\begin{array}{c}
a[0][0] \cdots a[0][C-1] \\
\vdots \\
a[R-1][0] \cdots a[R-1][C-1]
\end{array}
\]
Nested Array Row Access

Row Vectors

- \( A[i] \) is array of \( C \) elements
- Each element of type \( T \)
- Starting address \( A + i \times C \times K \)

```c
int A[R][C];
```

```plaintext
A[0]  \ldots  A[0][C-1]
\uparrow
A

A[i]  \ldots  A[i][C-1]
\uparrow
A+i\times C*4

A[R-1]  \ldots  A[R-1][C-1]
\uparrow
A+(R-1)\times C*4
```
**Nested Array Row Access Code**

```c
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

**Row Vector**
- `pgh[index]` is array of 5 int's
- Starting address `pgh + 20 * index`

**Code**
- Computes and returns address
- Compute as `pgh + 4 * (index + 4 * index)`

```asm
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax # pgh + (20 * index)
```
Nested Array Element Access

Array Elements

- $A[i][j]$ is element of type $T$
- Address $A + (i \cdot C + j) \cdot K$

```
int A[R][C];
```

```
A[0]

A[0][0]  ...  A[0][C-1]
```

```
A+i*C*4

A+(i*C+j)*4
```

```
A[i]

...  A[i][j]  ...
```

```
A[R-1]

A[R-1][0]  ...  A[R-1][C-1]
```

```
A+(R-1)*C*4
```
Nested Array Element Access Code

Array Elements

- \( pgh[index][dig] \) is int
- Address:
  \[ pgh + 20*index + 4*dig \]

Code

- Computes address
  \[ pgh + 4*dig + 4*(index+4*index) \]
- `movl` performs memory reference

```c
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```assembly
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx       # 4*dig
leal (%eax,%eax,4),%eax    # 5*index
movl pgh(%edx,%eax,4),%eax # *(pgh + 4*dig + 20*index)
```
Strange Referencing Examples

```c
zip_dig
pgh[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][1-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of int's

```c
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 }
zip_dig mit = { 0, 2, 1, 3, 9 }
zip_dig ucb = { 9, 4, 7, 2, 0 }
```
Accessing Element in Multi-Level Array

Computation

- Element access
  \[ \text{Mem[Mem[univ+4*index]+4*dig]} \]
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

```c
int get_univ_digit
    (int index, int dig)
{
    return univ[index][dig];
}
```

```
# %ecx = index
# %eax = dig
leal 0(%ecx,4),%edx  # 4*index
movl univ(%edx),%edx  # Mem[univ+4*index]
movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```
Strange Referencing Examples

Reference | Address | Value | Guaranteed?
--- | --- | --- | ---
univ[2][3] 56+4*3 = 68 | 2 | Yes
univ[1][5] 16+4*5 = 36 | 0 | No
univ[2][-1] 56+4*-1 = 52 | 9 | No
univ[3][-1] ?? | ?? | No
univ[1][12] 16+4*12 = 64 | 7 | No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
Using Nested Arrays

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

**Limitation**
- Only works if have fixed array size

```c
#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;
}
```

Row-wise A
Column-wise B

(i,*)  (*)(,k)
Dynamic Nested Arrays

Strength

• Can create matrix of arbitrary size

Programming

• Must do index computation explicitly

Performance

• Accessing single element costly
• Must do multiplication

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

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

```
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)]
```
Dynamic Array Multiplication

Without Optimizations

- **Multiplies**
  - 2 for subscripts
  - 1 for data

- **Adds**
  - 4 for array indexing
  - 1 for loop index
  - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele
    (int *a, int *b,
     int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```
Optimizing Dynamic Array Multiplication

Optimizations
• Performed when set optimization level to -O2

Code Motion
• Expression \( i*n \) can be computed outside loop

Strength Reduction
• Incrementing \( j \) has effect of incrementing \( j*n+k \) by \( n \)

Performance
• Compiler can optimize regular access patterns

```
{ 
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}

{ 
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}
```
Structures

Concept

• Contiguously-allocated region of memory
• Refer to members within structure by names
• Members may be of different types

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

Accessing Structure Member

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Structure Member

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

Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```c
int * find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```assembly
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Structure Referencing (Cont.)

C Code

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

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

Element i

<table>
<thead>
<tr>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

# %edx = r
movl (%edx),%ecx # r->i
leal 0(%ecx,4),%eax # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx) # Update r->p
Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

Compiler

- Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment

Size of Primitive Data Type:

- **1 byte** (e.g., char)
  - no restrictions on address
- **2 bytes** (e.g., short)
  - lowest 1 bit of address must be 0₂
- **4 bytes** (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00₂
- **8 bytes** (e.g., double)
  - Windows (and most other OS’s & instruction sets):
    » lowest 3 bits of address must be 000₂
  - Linux:
    » lowest 2 bits of address must be 00₂
    » i.e., treated the same as a 4-byte primitive data type
- **12 bytes** (long double)
  - Linux:
    » lowest 2 bits of address must be 00₂
    » i.e., treated the same as a 4-byte primitive data type
Satisfying Alignment with Structures

Offsets Within Structure
• Must satisfy element’s alignment requirement

Overall Structure Placement
• Each structure has alignment requirement K
  – Largest alignment of any element
• Initial address & structure length must be multiples of K

Example (under Windows):
• K = 8, due to double element

struct S1 {
  char c;
  int i[2];
  double v;
} *p;
Linux vs. Windows

Windows (including Cygwin):
• $K = 8$, due to double element

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

Linux:
• $K = 4$; double treated like a 4-byte data type
Effect of Overall Alignment Requirement

struct S2 {
    double x;
    int i[2];
    char c;
} *p;

\[ p \text{ must be multiple of:} \]
\[ 8 \text{ for Windows} \]
\[ 4 \text{ for Linux} \]

struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;

\[ p \text{ must be multiple of 4 (in either OS)} \]
Ordering Elements Within Structure

```c
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

10 bytes wasted space in Windows

2 bytes wasted space
Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Accessing Element within Array

• **Compute offset to start of structure**
  – Compute $12i$ as $4(i+2i)$

• **Access element according to its offset within structure**
  – Offset by 8
  – Assembler gives displacement as $a + 8$
    » Linker must set actual value

```c
short get_j(int idx)
{
    return a[idx].j;
}
```

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

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - `a` must be multiple of 4
- Offset of element within structure must be multiple of element’s alignment requirement
  - `v`’s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Union Allocation

Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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

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

(Windows alignment)
Using Union to Access Bit Patterns

```c
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;
}
```

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`
Byte Ordering

Idea
- Long/quad words stored in memory as 4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian
- Most significant byte has lowest address
- IBM 360/370, Motorola 68K, Sparc

Little Endian
- Least significant byte has lowest address
- Intel x86, Digital VAX
Byte Ordering Example

union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>c1</td>
<td>c2</td>
<td>c3</td>
<td>c4</td>
<td>c5</td>
<td>c6</td>
<td>c7</td>
</tr>
<tr>
<td>s0</td>
<td>s1</td>
<td>s2</td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i0</td>
<td>i1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```
Byte Ordering on x86

Little Endian

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]

Output on Pentium:
Byte Ordering on Sun

Big Endian

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{f0} & \text{f1} & \text{f2} & \text{f3} & \text{f4} & \text{f5} & \text{f6} & \text{f7} \\
\hline
\text{c[0]} & \text{c[1]} & \text{c[2]} & \text{c[3]} & \text{c[4]} & \text{c[5]} & \text{c[6]} & \text{c[7]} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} \\
\hline
\text{s[0]} & \text{s[1]} & \text{s[2]} & \text{s[3]} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} \\
\hline
\text{i[0]} & \text{i[1]} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{MSB} & \text{LSB} \\
\hline
\text{l[0]} \\
\end{array}
\]

Output on Sun:

- **Characters** 0–7 == \([0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]\)
- **Shorts** 0–3 == \([0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]\)
- **Ints** 0–1 == \([0xf0f1f2f3, 0xf4f5f6f7]\)
- **Long** 0 == \([0xf0f1f2f3]\)
Byte Ordering on Alpha

Little Endian

Output on Alpha:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]
Summary

Arrays in C
- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations
- Compiler often turns array code into pointer code
  \[ \text{zd2int} \]
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures
- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions
- Overlay declarations
- Way to circumvent type system