Agenda

- Symbol tables
- Semantic analysis
Symbol tables

- An *identifier* is a language token type, while the value of an identifier is a *name* (typically a string)

- A *symbol table* maps *names* to *symbols*

- What language constructs create new *names*, and when are those names visible in subsequent parts of the program text?

- What information is useful to track for each symbol?

- Each language will have its own rules about scoping and symbol visibility:
  - Hierarchical
  - Parallel
Symbol table entry

- Original point of instantiation
- Kind: constant, variable, type, procedure/function
- Type information:
  - Scalar vs array vs routine
  - Link to record/class/etc. definition
  - For routines: formal parameters and their symbol information, optional return type
- Language-specific attributes
- Storage size of item
- Runtime address (offset within stack)
- Visibility modifiers
- Uses (optional)
Implementing a symbol table

• Important operations:
  • Create a nested scope
  • Exit from a scope
  • Lookup name in the current scope
  • Lookup name according to language scoping rules
  • Put a new name-to-symbol entry in the table

• Hierarchical map of names to symbols
  • Stack of hash tables
  • Or, a hash table of lists
Implementing a symbol table

- Entering into a major or minor scope will create a nested symbol table
  - Maps naturally between enter & exit and push & pop
- If you perform an Ident-to-Symbol AST transformation, do you still need the hash table anymore?
  - Debuggers depend on knowing what names are visible/accessible at each point in the program
- Create a new name-to-symbols map anywhere new identifiers can be introduced
float gpa;

int main(int argc, char *argv[])
{
    char *course;
}

<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpa</td>
<td>var</td>
<td>float</td>
</tr>
<tr>
<td>main</td>
<td>func</td>
<td>...</td>
</tr>
<tr>
<td>argc</td>
<td>param</td>
<td>int</td>
</tr>
<tr>
<td>argv</td>
<td>param</td>
<td>char **</td>
</tr>
<tr>
<td>course</td>
<td>var</td>
<td>char *</td>
</tr>
</tbody>
</table>
Type tables

- Languages that support user-defined types require additional name-to-type mapping tables

- Scoping rules may dictate whether these names appear in parallel scopes, or alongside other kinds of symbols
Type table entries

• Typical details:
  • Name
  • Kind: struct, union, enum, typedef, scalar
  • Storage size
  • Runtime information
int t;

struct S {
    char *name;
    int number;
};
int t;

struct S {
  char *name;
  int number;
};
int t;

struct S {
    char *name;
    int number;
};

typedef struct S R;
C type table design
Memory layout

- A lower level language like C exposes the programmer to certain machine & memory characteristics
- Types have a natural size and typically must obey certain machine alignment restrictions
- struct’s and union’s inherit the most restrictive alignment of their members
```c
struct {
  u8  a;
  u16 b;
  u32 c;
  u64 d;
};
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```c
struct {
    u8 a;
    u16 b;
    u32 c;
    u64 d;
};
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
struct {
    u8 a;
    u16 b;
    u32 c;
    u64 d;
};
```c
struct {
    u8 a;
    u16 b;
    u32 c;
    u64 d;
};
```
```c
struct {
    u8 a;
    u16 b;
    u32 c;
    u64 d;
};
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>
struct {
    u8 a;
    u16 b;
    u32 c;
    u64 d;
};

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

16 bytes required to store 15 bytes of information

struct alignment = 8 bytes
```c
struct {
    u8 a;
    u64 d;
    u16 b;
    u32 c;
};
```
struct {
    u8 a;
    u64 d;
    u16 b;
    u32 c;
};

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
| a |   |   |   |   |   |   |   |   |   |   |   |   |   |   | d |
```
```c
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```
```c
struct {
    u8 a;
    u64 d;
    u16 b;
    u32 c;
};
```
struct {
    u8 a;
    u64 d;
    u16 b;
    u32 c;
};

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>20</td>
</tr>
</tbody>
</table>

24 bytes required to store 15 bytes of information
struct alignment = 8 bytes
Semantic analysis
Semantic analysis

- Checking and enforcement of non-syntactic language constraints
  - During compilation: static analysis
  - At runtime: dynamic analysis

- Kinds of analysis:
  - Visibility and accessibility
  - Type checking
  - Proper usage
  - Escape
  - Range

- Symbol tables typically construct during these analyses
## Semantic analysis example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>=</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visibility</strong></td>
<td>declared(A)?</td>
<td></td>
<td>declared(B)?</td>
</tr>
<tr>
<td></td>
<td>visible(A)?</td>
<td></td>
<td>visible(B)?</td>
</tr>
<tr>
<td><strong>Accessibility</strong></td>
<td>access(A)?</td>
<td></td>
<td>access(B)?</td>
</tr>
<tr>
<td></td>
<td>write(A)?</td>
<td></td>
<td>read(B)?</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>variable(A)?</td>
<td></td>
<td>variable(B)?</td>
</tr>
<tr>
<td></td>
<td>const(B)?</td>
<td></td>
<td>function(B)?</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>type(A)?</td>
<td></td>
<td>type(B)?</td>
</tr>
<tr>
<td></td>
<td>type(B)?</td>
<td></td>
<td>type(B)?</td>
</tr>
<tr>
<td></td>
<td>type(B)?</td>
<td></td>
<td>type(B)?</td>
</tr>
<tr>
<td></td>
<td>scalar(A)?</td>
<td></td>
<td>scalar(B)?</td>
</tr>
<tr>
<td></td>
<td>scalar(B)?</td>
<td></td>
<td>scalar(B)?</td>
</tr>
<tr>
<td></td>
<td>scalar(B)?</td>
<td></td>
<td>scalar(B)?</td>
</tr>
<tr>
<td></td>
<td>params(B)?</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td></td>
<td></td>
<td>assignTo(A, B)?</td>
</tr>
</tbody>
</table>
Type equivalence, compatibility and suitability

• When different types can be used together depends on the context and is governed by language specific rules

• **Type equivalence**: when can two objects of two different types be considered *equivalent*

• **Type compatibility**: when can two objects of two different types be considered *compatible*

• **Type suitability**: when can two objects of two different types be considered suitable for one another

• Assignment usually requires compatibility, expression operands usually require suitability
Type equivalence rules

• Name type equivalence:
  • Two types are *name equivalent* if they derive from the same definition
  • Allows for aliases such as *typedef's*

• Structural type equivalence:
  • Two types are *structural equivalent* if their definitions line up with one another (same structure, same values, same types)
Name type equivalence

struct S { int foo; };
typedef struct S A;
typedef struct S B;

struct S, A and B are all *name type equivalent*
Structural equivalence

typedef struct {
    int   a;
    char *b;
    float c;
} X;

typedef struct {
    int   p;
    char *q;
    float r;
} Y;

X and Y are structurally equivalent
Type equivalence rules

- Type equivalence checking is used to ensure that pointers match the data type they are pointing to
  - When a pointer is assigned the address-of something
  - When a variable is passed by-reference as a parameter
- Type equivalence implies *memory layout equivalence*
Go: structural copying

type X struct {
    F string
    G int
}

type Y struct {
    F string
    G int
}

var x X
var y Y

... y = Y(x)
Type checking
Type this expression

1 < 2
Type this expression

0 < True
Type this expression

\((0 < \text{True}) :: \text{boolean}\)
Type *check* this expression

1 < 2
Type **check** this expression

$$0 < True$$
Type **check** this expression

$0 < \text{True}$
Type this expression

\[ f(x, y, z) \]
Type *check* this expression

\[ f(x, y, z) \]
Type & type check this expression

\[ f(x, y, z) \]

func f(a, b, c integer) boolean