COEN 175

Lecture 8: Data Type Modeling

Types in Simple C

- We don't want to start writing C++ code just yet.
- Instead, we want to identify what information a type should hold.
- In other words, we need the requirements.
- A type consists of a **specifier** and **declarators**.
  - A specifier is one of `int`, `char`, or `double`.
  - A declarator can be a pointer, array, or function.
- In what ways can these be combined?

Examples

- Which of these declarations are legal in Simple C?

```
int x;              // legal: int
double *p;          // legal: pointer to double
char a[10];         // legal: array of char
int **q;            // legal: pointer to pointer to int
double *b[10];      // legal: array of pointers to double
char (*c)[10];      // illegal: pointer to array of char
int **d[10];        // legal: array of pointers to pointers
int *(*e)[10];      // illegal: pointer to array of pointers
```

More Examples

- Which of these declarations are legal in Simple C?

```
int f(int x);          // legal: func returning int
double *g(void);       // legal: func returning ptr to double
char a(void)[10];      // illegal (also illegal in C)
int b[10](void);       // illegal (also illegal in C)
double **h(void);      // legal: func returning ptr to ptr
char (*p)(int x);      // illegal: ptr to func returning char
int *(*q)(int x);      // illegal: ptr to func returning ptr
int f();               // illegal: Simple C doesn’t allow it!
```

Types in Simple C

- Pointer declarators are always allowed, but always have the lowest precedence:
  - Array of pointers, not pointer to array;
  - Function returning pointer, not pointer to function.
- At most one array or function declarator is allowed, and it always has the highest precedence.
- We can choose to model types as a triple:
  - Specifier;
  - Indirection (number of pointers);
  - Kind: function, array, or scalar (no declarators).

Examples

- What are the triples for these declarations?

```
int x;                // (INT, 0, SCALAR)
double *p;            // (DOUBLE, 1, SCALAR)
char a[10];           // (CHAR, 0, ARRAY)
int **q;              // (INT, 2, SCALAR)
double *b[10];        // (DOUBLE, 1, ARRAY)
int **d[10];          // (INT, 2, ARRAY)
int f(int x);         // (INT, 0, FUNCTION)
double *g(int x);     // (DOUBLE, 1, FUNCTION)
```

Computing Information

- Should we start writing C++ code yet? No.
  - We still need to make sure we can gather the necessary information.
  - No point in writing a class if we can't test it!
- Let's modify our parser to compute the necessary information.
  - We'll need to pass information between rules.
  - To do that we will need to use inherited and synthesized attributes.

What Rules Are Affected?

- Let's look at the relevant grammar rules:

$$
\begin{array}{lcl}
\textit{declaration} & \rightarrow & \textit{specifier declarator-list} \ \mathbf{;} \\
\textit{declarator-list} & \rightarrow & \textit{declarator} \\
 & | & \textit{declarator} \ \mathbf{,} \ \textit{declarator-list} \\
\textit{declarator} & \rightarrow & \textit{pointers} \ \mathbf{id} \\
 & | & \textit{pointers} \ \mathbf{id} \ [\ \mathbf{integer}\ ] \\
\textit{pointers} & \rightarrow & \epsilon \\
 & | & * \ \textit{pointers}
\end{array}
$$

- All the actions will take place in *declarator*.
  - Pointer declarators will be a **synthesized** attribute.
  - The specifier will be an **inherited** attribute.

Computing Indirection

- Let's modify the code for `pointers()` to compute the number of levels of indirection.

```
// modified code

unsigned pointers() {
    unsigned count = 0;

    while (lookahead == '*') {
        match('*');
        count ++;
    }

    return count;
}
```

Computing the Specifier

- Let's modify the code for `specifier()` to return the type specifier.

```
// modified code

int specifier() {
    int typespec = lookahead;

    if (lookahead == INT)
        match(INT);
    else if (lookahead == DOUBLE)
        match(DOUBLE);
    else
        match(CHAR);

    return typespec;
}
```

Inheriting the Specifier

- Let's modify the code for `declaration()` to pass in the type specifier.

```
// modified code

void declaration() {
    int typespec = specifier();
    declarator(typespec);

    while (lookahead == ',') {
        match(',');
        declarator(typespec);
    }

    match(';');
}
```

Inheriting the Specifier

- Finally, let's modify the code for `declarator()` to accept the inherited attribute.

```
// modified code

void declarator(int typespec) {
    unsigned indirection = pointers();
    match(ID);

    if (lookahead == '[') {
        match('[');
        match(INTEGER);
        match(']');
        cout << "(" << typespec … << ", ARRAY)" << endl;
    } else
        cout << "(" << typespec … << ", SCALAR)" << endl;
}
```

Missing Information

- Our requirements analysis of types in Simple C has left out two important pieces of information:
  - Array length
  - Function parameters
- Clearly, we can model these as follows:
  - Array length is simply an unsigned value.
  - Function parameters are a list (i.e., vector) of types.
  - We can either create separate C++ types (i.e., subclasses) for arrays and functions, or just store them all as one C++ type since the overhead is minimal.