MODERN C++ DESIGN PATTERNS

Unions: A Space-Saving Class

by Prasanna Ghali

union

- Similar to structure or class but one major difference that relates to memory layout …
- All union members have offset of zero
  - Storage of individual members is thus overlaid: only one member at time can be stored there
- Typically used to save space, by not storing all possibilities for certain data items that cannot occur together

Space Saving Structure (1/3)

- Parser stores information about each symbol in expression using `struct`
- Each symbol requires ?? bytes

```
struct Symbol {
  enum {
     OPERATOR, INTEGER, FLOAT, IDENTIFIER
  } kind;
  int8_t    op;
  int8_t    id;
  int32_t ival;
  float   fval;
};
```

Space Saving Structure (2/3)

- Expression A + 23 * 3.14 stored like this

```
struct Symbol sym1, sym2, sym3, sym4, sym5;
sym1.kind = IDENTIFIER;
sym1.id = 'A';
sym2.kind = OPERATOR;
sym2.op = '+';
sym3.kind = INTEGER;
sym3.ival = 23;
sym4.kind = OPERATOR;
sym4.op = '*';
sym5.kind = FLOAT;
sym5.fval = 3.14F;
```

Space Saving Structure (3/3)

- `union` is better solution to store mutually exclusive data members
- Each symbol now requires 8 bytes

```
union USymbol {
  int8_t  op;
  int8_t  id;
  int32_t ival;
  float   fval;
};
```

```
struct Symbol {
  enum {
     OPERATOR, INTEGER, FLOAT, IDENTIFIER
  } kind;
  USymbol data;
};
```

Initializing Unions: Caveat

- Type of initializer must be same type as first member of union!!!

```
union USymbol {
  int8_t  op;
  int8_t  id;
  int32_t ival;
  float   fval;
};
```

```
struct NewSymbol {
  enum {
     OPERATOR, INTEGER, FLOAT, IDENTIFIER
  } kind;
  USymbol data;
};
```

```
// fine, op is first member
struct NewSymbol sym1 = {NewSymbol::OPERATOR, {'+'} };

// error, narrowing of float to int8_t
struct NewSymbol sym2 = {NewSymbol::FLOAT, {3.14} };
```

Space Saving Structure: Another Example (1/2)

- Input event from user consists of *either* keyboard press or mouse move …

```
// mouse: movement
struct Mouse {
  int32_t x, y;
};

// keyboard: press/unpress of key
using KeyCode = int32_t;

// which type of input?
enum class EventType {
  MouseMove, KeyPress
};
```

```
union EventData {
  Mouse   mouse;
  KeyCode key;
};

struct Event {
  EventType event_type;
  EventData event_data;
};
```

Space Saving Structure: Another Example (2/2)

- Even better …

```
struct Event {
  using KeyCode = int32_t;

  enum { MouseMove, KeyPress } EventType;

  union {
    struct {
      int32_t x, y;
    } mouse;
    KeyCode key_code;
  };
};
```

union for Convenient Access


```
union Color {
  uint32_t rgbi;
  uint8_t  rgba[4];
  struct {
    uint8_t pad;
    uint8_t blue;
    uint8_t green;
    uint8_t red;
  };
};
```

```
union vector3 {
  float c[3];
  struct {
    float x, y, z;
  };
};
```

Anonymous structures are prohibited in ISO C++!!!

Solution: Remove `-pedantic-errors` flag!!!

union for Different Interpretations


```
union FloatInt {
  float    f;
  uint32_t i;
};

// IEEE 754 format uses bits [30,23] for exponent
uint32_t Exponent(FloatInt fi) {
  return (fi.i >> 23) & 0x00ff;
}

// IEEE 754 format uses bits [22, 0] for mantissa
uint32_t Mantissa(FloatInt fi) {
  return fi.i & ((1 << 23) - 1);
}
```

unions are Simple …

- Restrictions for anonymous unions
  - Cannot have anonymous structures
  - Cannot have member functions
  - Cannot have static data members
  - Cannot have non-public data members
- Restrictions for unions:
  - Cannot have inheritance relationships nor virtual functions
  - Cannot have non-static data member of reference types
  - Objects don't know which type of value they currently hold and therefore, you cannot have non-trivial members without extra effort

C++17 std::variant<>

- Provides *closed discriminated union* (which means there is specified list of possible types)
  - Where type of current value is always known
  - That can hold values of any specified type
  - That you can *derive* from

C++17 std::variant<>


```cpp
#include <variant>

std::variant<int, double> v, w;
v = 123; // activate v<int>
try {
  int i = std::get<int>(v); // i is 123
  int j = std::get<0>(v);  // j is also 123
  w = std::get<int>(v); // w's int is now activated
  v = 456.789; // v<double> is set
  w = v; // w is equivalent to v
  std::cout << "w<int>: " << std::get<int>(w) << "\n";
} catch(std::bad_variant_access const& e) {
  std::cout << e.what() << "\n";
}
```

C++17 std::variant<>


```
std::variant<std::vector<int>, std::string> vs;
vs = "hello world";
std::cout << "which member set: "
          << vs.index() << "\n";
std::cout << std::get<std::string>(vs) << "\n";

vs = std::vector<int>{1,2,3,4};
std::cout << std::get<std::vector<int>>(vs) << "\n";
```