EDAF50 – C++ Programming

11.Low-level details. Loose ends.

Sven Gestegård Robertz

Computer Science, LTH

2018

Outline

union

The size of a normal **struct** (**class**) is *the sum* of its members

```
struct DataS {
    int nr;
    double v;
    char txt[6];
};
```

All members in a **struct** are laid out after each other in memory.

The size of a **union** is equal to *the size of the largest member*

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

All members in a **union** have *the same address*: only one member can be used at any time.

union

Example use of `DataU`

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

```
DataU a;

a.nr = 57;
cout << a.nr << endl;            57

a.v = 12.345;
cout << a.v << endl;             12.345

strncpy(a.txt, "Tjo",5);
cout << a.txt << endl;           Tjo
```

The programmer is responsible for only using the "right" member

union
Example of wrong use

```
using std::cout;
using std::endl;

union Foo{
  int i;
  float f;
  double d;
  char c[10];
};
int main()
{
  Foo f;

  f.i = 12;
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;

  strncpy(f.c, "Hej, du", 9);
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;
}
```

```
12, 1.68156e-44, 5.92879e-323, ^L
745170248, 3.33096e-12, 1.90387e-306, Hej, du
```


union
encapsulate a union in a class to reduce the risk of mistakes

```
struct Bar{
    enum {undef, i, f, d, c} kind;
    Foo u;
};
void print(Bar b) {
    switch(b.kind){
    case Bar::i:
        cout << b.u.i << endl;
        break;
    case Bar::f:
        cout << b.u.f << endl;
        break;
    case Bar::d:
        cout << b.u.d << endl;
        break;
    case Bar::c:
        cout << b.u.c << endl;
        break;
    default:
        cout << "???" << endl;
        break;
    }
}
```

```
void test_kind()
{
    Bar b{};

    b.kind = Bar::i;
    b.u.i = 17;

    print(b);

    Bar b2{};
    print(b2);
}
```

```
17
???
```

union
anonymous `union` – removes one level

An alternative to the previous example:

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
};

FooS test;

test.kind = FooS::k_c;
strncpy(test.c, "Testing", 9);
if(test.kind == FooS::k_c)
    cout << test.c << endl;
```

Exposing the tag to the users is brittle.

```
Testing
```


Tagged union
A class with anonymous `union` and access functions

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
  FooS()           :kind{undef} {}
  FooS(int ii)     :kind{k_i},i{ii} {}
  FooS(float fi)   :kind{k_f},f{fi} {}
  FooS(double di)  :kind{k_d},d{di} {}
  FooS(const char* ci) :kind{k_c} {strncpy(c,ci,10);}
  int    get_i() {assert(kind==k_i); return i;}
  float  get_f() {assert(kind==k_f); return f;}
  double get_d() {assert(kind==k_d); return d;}
  char*  get_c() {assert(kind==k_c); return c;}
  FooS& operator=(int ii)     {kind=k_i; i = ii; return *this;}
  FooS& operator=(float fi)   {kind=k_f; f = fi; return *this;}
  FooS& operator=(double di)  {kind=k_d; d = di; return *this;}
  FooS& operator=(const char* ci){kind=k_c; strncpy(c,ci,10);
                                  return *this;}
};
```


Bitwise operators

Bitwise and: `a & b`
Bitwise or: `a | b`
Bitwise xor: `a ^ b`
Bitwise complement: `~a`

shift left: `a << 5`
shift right: `a >> 5`

>> on signed types is implementation defined

Common operations:

set 5th bit
```
a = a | (1 << 4);
a |= (1 << 4);
```

clear 5th bit
```
a = a & ~(1 << 4);
a &= ~(1 << 4);
```

toggle 5th bit
```
a = a ^ (1 << 4);
a ^= (1 << 4);
```


Bitwise operators
Example:

Low-level operations: Bitwise operators
All variables are unsigned 16 bit integers

```
a = 77;           // a = 0000 0000 0100 1101
b = 22;           // b = 0000 0000 0001 0110
c = ~a;           // c = 1111 1111 1011 0010
d = a & b;        // d = 0000 0000 0000 0100
e = a | b;        // e = 0000 0000 0101 1111
f = a ^ b;        // f = 0000 0000 0101 1011
g = a << 3;       // g = 0000 0010 0110 1000
h = c >> 5;       // h = 0000 0111 1111 1101
i = a & 0x000f;   // i = 0000 0000 0000 1101
j = a | 0xf000;   // j = 1111 0000 0100 1101
k = a ^ (1 << 4); // k = 0000 0000 0101 1101
```


Bit-fields
Can be used to save memory

Specify explicit size in bits with `var : bit_width`

```
struct Car {// record in a car database
    char reg_nr[6];  NB! not null-terminated.
    unsigned int model_year : 12;
    unsigned int tax_paid : 1;
    unsigned int inspected : 1;
};
```

`sizeof(Car) = 8` *on my computer*

Bit-fields
Example

Access of members

```
Car c;

strncpy(c.reg_nr, "ABC123", 6);
c.model_year = 2011;
c.tax_paid = true;
c.inspected = true;

cout << "Year: " << c.model_year << endl;
if (c.tax_paid && c.inspected)
    cout << std::string(c.reg_nr, c.reg_nr+6) << " is OK";
```

Bit-fields
Warnings

Bit-fields can be useful in special cases, but they are *not portable*

- the layout of the object is *implementation defined*
- the compiler can add *padding*
- bit-field members *have no address*
  - cannot use the address-of operator &
- always specify **signed** or **unsigned**
  - use **unsigned** for members of size 1
- access can be slower than a "normal" struct
- integer variables and bitwise operations is an alternative

std::bitset (<bitset>)

- efficient class for storing a set of bits
  - compact
  - fast
- has convenient functions
  - test, **operator**[]
  - set, reset, flip
  - any, all, none, count
  - conversion to/from string
  - I/O operators
- cf. std::vector<**bool**>
  - std::bitset has fixed size
  - a std::vector can grow
  - but does not quite behave like a normal std::vector<T>

bitset
Example: store 50 flags in 8 bytes

```
void test_bitop(){
  bool status;
  cout << std::boolalpha;

  unsigned long quizA = 0;

  quizA |= 1UL << 27 ;
  status = quizA & (1UL << 27);
  cout << "student 27: " ;
  cout << status << endl;

  quizA &= ~(1UL << 27);
  status = quizA & (1UL << 27);
  cout << "student 27: ";
  cout << status << endl;
}
student 27: true
student 27: false
```

```
void test_bitset(){
  bool status;
  cout << std::boolalpha;

  std::bitset<50> quizB;

  quizB.set(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;

  quizB.reset(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;
}
student 27: true
student 27: false
```


The comma operator
(Introduction and warning)

The comma operator expression expression1, expression2

- First evaluates expression1, then expression2
- the expression has the value of expression2
- NB! The comma separating function parameters or arguments is *not* the comma operator
- Examples:

```
string s;
while(cin >> s, s.length() > 5) {
  //do something
}

std::vector<int> v(10);

vector<int>::size_type cnt = v.size();
for(vector<int>::size_type i=0; i < v.size(); ++i, --cnt) {
  v[i]=cnt;
}                    v now contains 10 9 8 7 6 5 4 3 2 1
```

Do not use the comma operator!

C-strings – library functions

functions in <cstring>

```
strcpy(dest,src)     // Copies src to dest
strncpy(dest,src,n)  // Copies at most n chars
                     NB! dest is not null-terminated when truncating

strcat(s,t)          // Appends a copy of t to the end of s
strncat(s,t,n)       // Appends at most n chars

strlen(s)            // Gives the length of s
strlen(s,n)          // Gives the length of s, max n chars

strcmp(s,t)          // Compare s and t
strncmp(s,t,n)       // ... at most n chars
// s<t, s==t, s>t returns <0, =0, >0 respectively
```

(even more) unsafe, avoid when possible!

Multidimensional arrays

multi-dimensional arrays

- Does not (really) exist in C++
  - are arrays of arrays
  - Look like in Java
- Java: array of *references to arrays*
- C++: two alternatives
  - Array of arrays
  - Array of *pointers* *(to the first element of an array)*

Multi-dimensional arrays

Initializing a matrix with an initializer list:

3 rows, 4 columns

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initializer list for row 0 */
  {4, 5, 6, 7} ,   /*  initializer list for row 1 */
  {8, 9, 10, 11}   /*  initializer list for row 2 */
};
```

Instead of nested lists one can write the initialization as a single list:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Multi-dimensional arrays are stored like an array in memory.
- The dimension *closest to the name* is the size of the array
- The remaining dimensions belong to the element type

Multi-dimensional arrays
Representation of arrays in memory

An array `T array[4]` is represented memory by a sequence of four elements of type T: | T | T | T | T |

An `int[4]` is represented as

| **int** | **int** | **int** | **int** |

Thus, `int[3][4]` is represented as

| **int** | **int** | **int** | **int** | **int** | **int** | **int** | **int** | **int** | **int** | **int** | **int** |

Multi-dimensional arrays

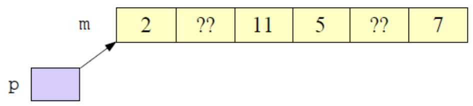
Examples

```
int m[2][3]; // A 2x3-matrix

m[1][0] = 5;

int* e = m;       // Error! cannot convert 'int [2][3]' to 'int*'
int* p = &m[0][0];
*p = 2;

p[2] = 11;
int* q=m[1];      // OK: int[3] decays to int*
q[2] = 7;
```



Multi-dimensional arrays

Parameters of type multi-dimensional arrays

```
// One way of declaring the parameter
void printmatr(int (*a)[4], int n);

// Another option
void printmatr(int a[][4], int n) {
{
   for (int i=0;i!=n;++i) {
      for (const auto& x : a[i]) { The elements of a are int[4]
         cout << x << " ";
      }
      cout << endl;
   }
}
```


Multi-dimensional arrays

Initialization and function call

```
int a[3][4] {1,2,3,4,5,6,7,8,9,10,11,12};
int b[3][4] {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

printmatr(a,3);
cout << "-----------------" << endl;
printmatr(b,3);
```

```
      1        2        3        4
      5        6        7        8
      9       10       11       12
-----------------
      1        2        3        4
      5        6        7        8
      9       10       11       12
```


Variadic templates

```
void println() { base case: no arguments
    cout << endl;
}

template <typename T, typename... Tail>
void println(const T& head, const Tail&... tail)
{
    cout << head << " ";    Print the first element
    println(tail...);       rekursion: print the rest
}

void test_variadic()
{
    string a{"Hello"};
    int b{10};
    double c{17.42};
    long d{100};

    println(a,b,c,d);

}
```

Template metaprogramming

- Write code that is executed *by the compiler, at compile-time*
- Common in the standard library
  - As optimization: move computations from run-time to compile-time
  - As utilities: e.g., type_traits, iterator_traits
- Metafunction: a class template containing the result
- Standard library conventions:
  - Type results: type member named type
  - Value results: value member named value

Template metaprogramming
Example of compile-time computation

```
template <int N>
struct Faculty{
    static constexpr int value = N * Faculty<N-1>::value;
};

template <>
struct Faculty<0>{
    static constexpr int value = 1;
};

void example()
{
    display<Faculty<5>::value>();
}
```

Result of the *meta-function call* as a compiler error:

```
tmp.cc: In instantiation of 'void tellme() [with T = int [120]]':
tmp.cc:9:31:   required from 'void display() [with int N = 120]'
tmp.cc:24:28:   required from here
tmp.cc:6:26: error: 'thetype' is not a member of 'int [120]'
```


Template metaprogramming
Example of templates for getting values as compiler errors

- Trick: use a template that doesn't compile to get information about the template parameters through a compiler error.
- Can be useful for debugging templates.
- To get the type parameter T (assuming that T has no member function thetype()):

```
template <typename T>
void tellme() {T::thetype();}
```

- To get the int parameter N:

```
template <int N>
void display() {tellme<int[N]>();}
```


Most vexing parse
Example 1

```
struct Foo {
    int x;
};

int main()
{
#ifdef ERROR1
    Foo f(); // function declaration
#else
    Foo f{}; // Variable declaration C++11
    // Foo f; //C++98 (but not initialized)
#endif
    cout << f.x << endl; // Error

    Foo g = Foo(); // OK     // C++11: auto g = Foo();
    cout << g.x << endl;
```

```
error: request for member 'x' in 'f', which is of
non-class type 'Foo()'
```


Most vexing parse
Example 2

```
struct Foo {
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

int main()
{
#ifdef ERROR2
    Bar b(Foo()); // function declaration
#else
    Bar b{Foo()}; // Variable declaration (C++11)
    // Bar b((Foo()));  // C++98 : extra parentheses --> expression
#endif
    cout << b.x << endl; // Error!
```

```
error: request for member 'x' in 'b', which is of
non-class type 'Bar(Foo (*)())'
```


Most vexing parse
Example: actual function

```
struct Foo {
    Foo(int i=0) :x{i} {}
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

Bar b(Foo()); // forward declaration

Foo make_foo()
{
   return Foo(17);
}

Bar b(Foo(*f)())
{
    return Bar(f());
}
```

```
void test()
{
    Bar tmp = b(make_foo);
    cout <<  tmp.x << endl;
}
```

```
17
```

Suggested reading

References to sections in Lippman

C-style strings	3.5.4
Multi-dimensional arrays	3.6
Bitwise operations	4.8
The comma operator	4.10
Union	19.6
Bit-fields	19.8.1