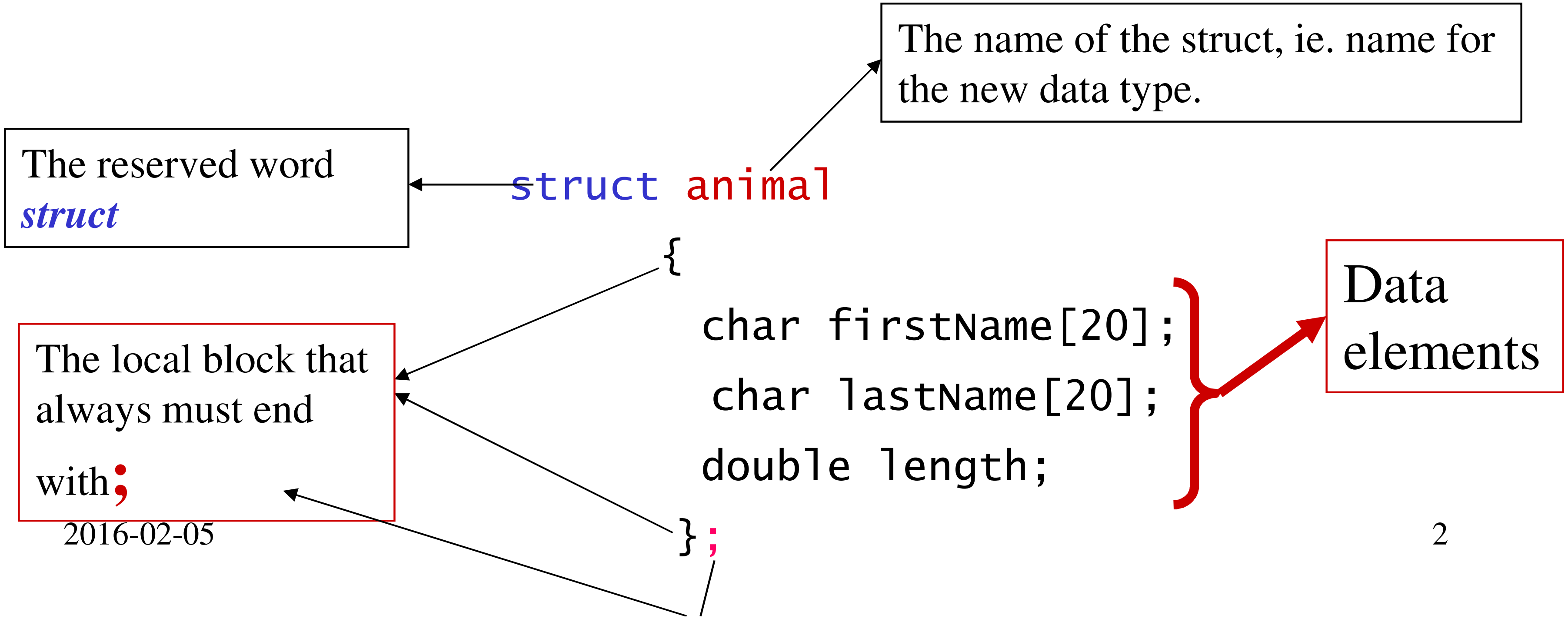
Struct, union and enum

What is a struct?

- A struct in its simplest form is a user defined data type that contains a collection of heterogeneous data elements.
- Data elements in a struct can be of the same type as those primitive data types (already defined) or other self-defined.



The name of the struct, ie. name for the new data type.

The reserved word *struct*

```
struct animal
     {
       char firstName[20];
       char lastName[20];
       double length;
     };
```

The local block that always must end with;

Data elements

Struct :another example: C++03

The collection of data elements belonging to the new data type often describe a thing /object.

```
struct date {
              int day;
              int month;
              int year;
         };

struct myCat {
                 char firstName[20];
                 char lastName[20];
                 double hight;
                 date birthDate;←——— struct in a struct
         };
```

struct

- In old C++03 one can not initialize members of a struct inside the definition, i.e. when creating the new data type.
- This should / can be done after definition of the strcut and then you have to first create a variable of the new data type, which means you initializes the members of that object (variable) and nothing else.

- In C++11 you can initialize members of a struct inside the definition.
- The new variable can be treated in the same way as other primitive data types, except that the new one has different members that must be accessed in other ways.

Example: initializing & Access (output)

```
int main()
{
  date bron;
  date death= { 01, 02, 2002 };
  myCat bear= {
                  ”John",
                  ”Bear",
                    23.5,
                   { 12, 12, 1900 }
                 };
   cout<< bear.firstName << " " << bear.lastName << " ”
       << bear.hight<< endl;
   cout<< "Årtal " << bear.birthDate.year << endl;
  cout<< "Dödsår " << death.year << endl;
```

Example (Cont.): Access (input)

```
cout << ”Type hight: ";
cin >> bear.hight;

cout << ”Type dd mm yyyy:";
cin >> bear.birthDate.day >> bear.birthDate.month
    >> bear.birthDate.year;

cout << ”Type firstname: ";
cin.getline(bear.firstName,20).get();

cout << ”Type lastname: ";
getline(cin,bear.lastName).get() ;
return 0;
}
```

Assignment

- You can assign new values, or the result of different expressions / calculations for individual elements to the struct variable.

```
born.day= 12/2;
born.day= death.day;
```

- One may also copy the contents of one variable to another, provided that they are of the same data type.

```
born = death;
```

arrays of struct

- One can create an array of struct in the same way as an array of int, char or other data types.

```
struct Animal {

char firstName[20];
char lastName[20];
double hight;

};
```

```
Animal mildadjur[3];
Animal vildadjur[ ]= {
                    { ”Minkatt",  "Troy", 26},
                    { ”Dinkatt” , "Michael", 25},
                    { ”Sovande” , ”Björn",   26}
                };

cout << vildadjur[0].firstName <<endl
     << vildadjur[0].hight << endl;

for (int i = 0; i <3; i++)
     vildadjur[i].hight=(rand()% 50) +1;
```

Sorting array of struct

- One can of course sort array of struct, based on ONE member, in this case hight

```
for (int i =0; i< 3; i++)
 for ( int j = 0; j < i; j++)
   if ( vildadjur[j].hight> vildadjur[i].hight )
         {
             animal temp = vildadjur[j];
             vildadjur[j]=vildadjur[i];
             vildadjur[i]= temp;
         }
```

Struct variabels as functionparameter

- Struct variables can be passed to functions, and these can also be returned.
- A function that has a struct as inparameter has the prototype: `void function( Date birthDate, int size);`
- The same function would have been able to return a variable of type Date, then the prototype like this:

  ```
  Date function( Date brithDate, int size);
  ```
- Another function prototype that has an array of structures as input parameter:

  ```
  void funktion( Animal vildadjur[], int size);
  ```

Example: Struct-variable as functionparameter

```
void sortera( Animal vildadjur[], int size)
{
  for (int i =0; i< size; i++)
   for ( int j = 0; j < i; j++)
     if ( vildadjur[j].hight > vildadjur[i].hight)
          {
             vildadjur temp = vildadjur[j];
             vildadjur[j]=vildadjur[i];
             vildadjur[i]= temp;
          }
}
```

- **functioncall:** `sortera (vildadjur,3);`

Union

- With a union you create a new type (like a struct) as a named part of the memory which at different times can hold objects of different types and sizes. Union provides a way to handle different types of data in a shared storage space, big enough to hold the largest of the types found in the Union.
- Declaration:

```
union MyLittleUnion
{
   char    aChar;
   int     anInt;
   double  aDouble;
};
```

How to use Union

```
int  main()
{
   MyLittleUnion aUnionObjekt;

   cout << "sizeof(aUnionObjekt) = "
        << sizeof(aUnionObjekt) << endl;

 aUnionObjekt.aChar = 'U';

 aUnionObjekt.anInt=32;

 aUnionObjekt.aDouble = 3234.1467899;

  cout << aUnionObjekt.aChar<<endl
       << aUnionObjekt.anInt<<endl
       << aUnionObjekt.aDouble<<endl;
  return 0;
}
```

Example with struct

```
struct Entry {
               char name[20];
               char type;
               char string_value[5];
               int  int_value;
              };
//string_value is used if type==‘s’
//int_value is used if type==‘i’
Entry p;
switch ( p.type)
{
  case ‘s’:  cout<< p.string_value<< endl; break;
  case ‘i’:  cout<< p.int_value<< endl;    break;
  default :  cerr<< “type corrupted”<< endl;
}
```

Example with union

Since **string_value** and **int_value** will not be used simultaneously, we can use the union instead, this way could some space!

```
struct Entry {
               char name[20];
               char type;
               union {
               char string_value[5];
               int  int_value;
               } u;
             };
Entry p;
switch ( p.type)
{
  case ‘s’:  cout<< p.u.string_value<< endl; break;
  case ‘i’:  cout<< p.u.int_value<< endl;    break;
  default :  cerr<< “type corrupted”<< endl; break;
}
```

Enum example:

- Declaration:

```
enum signal{off, on};
enum {lazy, hazy, crazy} why;
```

- Here I have created an enumerable type named signal.

- One can create variables of type signal in the same way as other primitive types int, char, etc.
- Declaration and initialization:
-
```
signal a;
a = on;
```

- **If you in an enumerable type use one name, you may not use that name again in another enumerable type.**
- This declaration:

```
enum answer {no, yes, maybe = -1};
```

is all ok, but not below, because no and yes is already in use.

```
enum negative {no,yes, minus} c;  //wrong
```

Enum example:

- **Given: int i;**

  The assignment i=a; is ok, as in this case we get a value of 1 (see last picture).

- **Given: answer b;**

  assignment b = a; is NOT ok, as a and b are of different types. (see last picture). But the assignment is accepted with a type conversion, ie. b = (answer) a; is ok.

- **Given why=hazy;**

  comparisons like:

  b = (why? no: yes);

  is ok, because (no, yes) is of the same type (answer).

  See last picture.

Enum example:

```
 enum game_result {WIN, LOSE, TIE, CANCEL};

for (int result= WIN ; result<= CANCEL ; result++)
 {
    if (result == CANCEL)
          cout << ” game suspended "<<endl;
      else
      {
         cout << ”we played”;
         if (resultat == WIN)
            cout << ”and we won!”;
         if (result == LOSE)
            cout << ”and we lost!!).”<<endl;
      }
   }
```

Enum example:

```
enum shape_type
{
  circle,
  square,
  rectangle
};

void main()
{
    shape_type shape = circle;

    switch(shape)
     {
       case circle:     cout<<circle; break;
       case square:     cout<<square; break;
       case rectangle:  cout<<rectangle; break;
    }
}
```

Enum Example:

```
//A little bit complex?!!

struct Sales
{
   enum WeekDay { Mon, Tues, Wed, Thu, Fri} ;
   int  sold[5];
 };

int main()
{
  Sales ThisWeek = { 23, 43, 12, 34, 32 };

  // Notice how I access WeekDay elements!
  //(Sales::Mon is //zero, next is 1 and so on.)

  for (int day = Sales::Mon; day <= Sales::Fri ; ++day)
    cout << ThisWeek.sold[day] <<endl;

  return 0;
}
```

Enumerated Types: Limitations

```
// this code won't compile!
enum Color {RED, GREEN, BLUE};
enum Feelings {EXCITED, MOODY, BLUE};
```

Enumerated Types: Limitations

```
enum Nail {RED, GREEN, BLUES};
enum Toothbrush {EXCITED, MOODY, BLUE};

 Nail yourNail=RED;
 Toothbrush myToothbrush=EXCITED;
  if (yourNail == myToothbrush)
              cout << "Yes thats like that." << endl;
 else
              cout << "Noway, they arent the same." << endl;
```

Output: ”Yes thats like that.”

Isn’t this strange? yourNail never should be myToothbrush

StrangeEnum

Enumerated Types: Limitations are away in C++11 by strongly typed enum

```
// This code WILL compile with C++11

enum class Color {RED, GREEN, BLUE};
enum class Feelings {EXCITED, MOODY, BLUE};
```

FixedEnum