4-algebraic-types

June 3, 2024

1 A brief introduction to the algebra of types

> In the functional world, building new types from old ones is usually done through two operations: sum and product (these new types are thus called algebraic). A product of two types A and B is a new type that contains an instance of A and an instance of B. (…) The sum of types A and B is a type that can hold an instance of A or an instance of B, but not both at the same time. *Ivan Cukic*

1.1 A * B * … : `std::tuple`

The parameterized class `std::tuple <A, B, ...>` is a generalization of `std::pair` to an arbitrary number of types. From the *type algebra* point of view, it can be considered as the multiplication of types.

1.1.1 Access by `std::get`

Different member variables of a tuple are accessible through their indices.

[1]:
```
#include <tuple>
#include <iostream>
#include <string>
#include <stdexcept>
```

[2]:
```
std::tuple<double, char, std::string> get_student( int id )
 {
  if (id == 0) return { 3.8, 'A', "Lisa Simpson" } ;
  if (id == 1) return { 2.9, 'C', "Milhouse Van Houten" } ;
  if (id == 2) return { 1.7, 'D', "Ralph Wiggum" } ;
  throw std::invalid_argument("unknown student id") ;
 }
```

[3]:
```
void display_student( int id )
 {
  auto student = get_student(id) ;

  std::cout
    << "GPA: " << std::get<0>(student) << ", "
    << "grade: " << std::get<1>(student) << ", "
    << "name: " << std::get<2>(student)
```

```
    << std::endl;
}
```

[4]:
```
display_student(0) ;
display_student(1) ;
display_student(2) ;
```

```
GPA: 3.8, grade: A, name: Lisa Simpson
GPA: 2.9, grade: C, name: Milhouse Van Houten
GPA: 1.7, grade: D, name: Ralph Wiggum
```

1.1.2 Access by "structured binding"

Since C ++ 17, we can "unpack" on the fly all kinds of structures, in order to distribute the values of the structure in a set of independent variables. In the example below, this allows us to give more meaningful names to the different elements of a `std::tuple` and to avoid the use of `std::get<>` :

[5]:
```
void display_student( int id )
{
 auto [ gpa, grade, name ] = get_student(id) ;

 std::cout
   << "GPA: " << gpa << ", "
   << "grade: " << grade << ", "
   << "name: " << name
   << std::endl;
}
```

[6]:
```
display_student(0) ;
display_student(1) ;
display_student(2) ;
```

```
GPA: 3.8, grade: A, name: Lisa Simpson
GPA: 2.9, grade: C, name: Milhouse Van Houten
GPA: 1.7, grade: D, name: Ralph Wiggum
```

Note: it is not necessarily intuitive, but it is complicated to loop through all the elements of a tuple. On this subject, see `std::apply`.

1.2 A + B + ... : `std::variant`

The parameterized class `std::variant<A, B, ...>` is an evolution of C unions. In addition, it memorizes the type of the last stored value and raises exceptions if the developer makes type errors. From the "type algebra" point of view, we can think of it as the addition of types.

1.2.1 Access when the current type is known: `std::get`

When we know at all times what the type of the current value of a variant, we can use `std::get<>` with the desired type, or even the type index:

[7]:
```
#include <iostream>
#include <variant>
```

[8]:
```
std::variant<int, double> v, w ;
int i, j ;
double x, y ;

v = 42 ;
w = v ;
i = std::get<int>(v) ;
j = std::get<0>(w) ;

w = 3.14 ;
v = w ;
x = std::get<double>(v) ;
y = std::get<1>(w) ;

// std::get<float>(v); // compilation error: no float in [int, double]
// std::get<3>(v);     // compilation error: valid index values are 0 and 1
// std::get<int>(v);   // runtime exception: v currently contains a double

std::cout<<i<<" "<<j<<" "<<x<<" "<<y<<" "<<std::endl ;
```

```
42 42 3.14 3.14
```

The advantage of `std::variant` here, compared to a union, is that a type mismatch between a write and a read is detected at runtime.

1.2.2 Access when the current type is unknown: `std::get_if`

If we do not know what is the type of the value currently stored in a variant, we will instead call `std::get_if` :

[9]:
```
#include <vector>
```

[10]:
```
using var_t = std::variant<int, double> ;
std::vector<var_t> vals = { 42, 3.14 } ;

for ( auto & val : vals )
 {
  int * ipval = std::get_if<int>(&val) ;
  double * dpval = std::get_if<double>(&val) ;

  if ( ipval ) std::cout << "int value: " << *ipval << std::endl ;
  else if ( dpval ) std::cout << "double value: " << *dpval << std::endl ;
  else throw "unknown value !" ;
 }
```

```
int value: 42
double value: 3.14
```

Note: as for the `std::tuple`, we have to know in advance all the types that make up our variant, and to try them all in the main program. This makes the program cumbersome and does not facilitate the extension of the variant with new types, since all the client codes will have to be updated. More complex syntax is available for writing more generic customer codes: see `std::visit`.

2 Questions ?

3 Exercise

In this example, replace inheritance with the use of a std::variant:

[1]:
```
%%file tmp.algebraic-types.cpp

#include <iostream>
#include <vector>
#include <memory>

struct Particle
 {
  virtual void print() = 0 ;
  virtual ~Particle() = default ;
 } ;

struct Electron : public Particle
 { virtual void print(){ std::cout<<"E"<<std::endl ; } } ;

struct Proton : public Particle
 { virtual void print(){ std::cout<<"P"<<std::endl ; } } ;

struct Neutron : public Particle
 { virtual void print(){ std::cout<<"N"<<std::endl ; } } ;

int main()
 {
  std::vector<std::unique_ptr<Particle>> ps ;
  ps.push_back(std::make_unique<Electron>()) ;
  ps.push_back(std::make_unique<Proton>()) ;
  ps.push_back(std::make_unique<Neutron>()) ;

  for ( auto & p : ps )
   { p->print() ; }
 }
```

```
Writing tmp.algebraic-types.cpp
```

[2]:
```
!rm -f tmp.algebraic-types.exe && g++ -std=c++17 tmp.algebraic-types.cpp -o tmp.
↪algebraic-types.exe
```

[3]:
```
!./tmp.algebraic-types.exe
```

```
E
P
N
```