TDDD38/726G82 - Advanced programming in C++

Sum Types in C++

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Intro

Goals

- C++ is statically typed

Intro

Goals

- C++ is statically typed
- Can we simulate dynamic typing though?

Intro

Type categories

Algebraic Data Types

- Product types

Intro

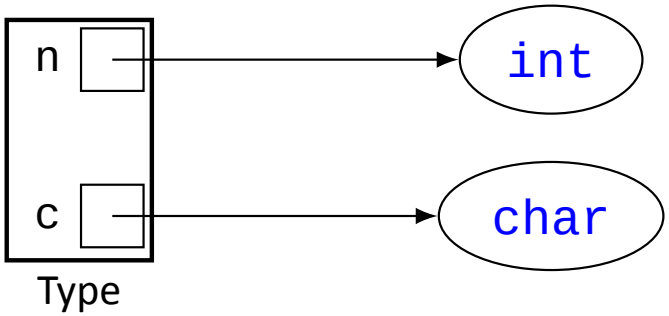
Type categories

Algebraic Data Types

- Product types
- Sum types

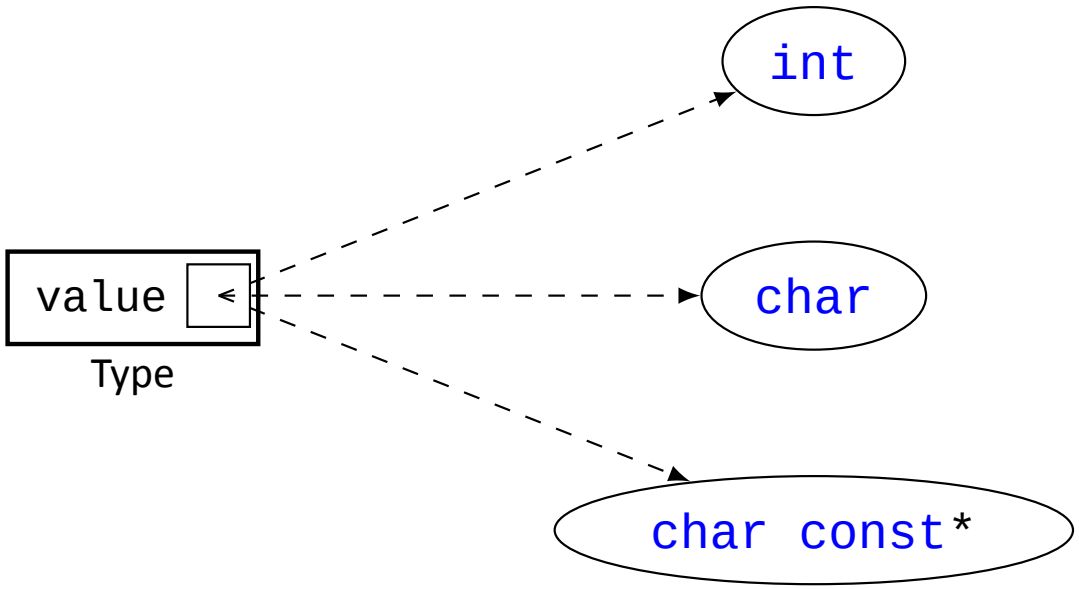
Intro

Product Type



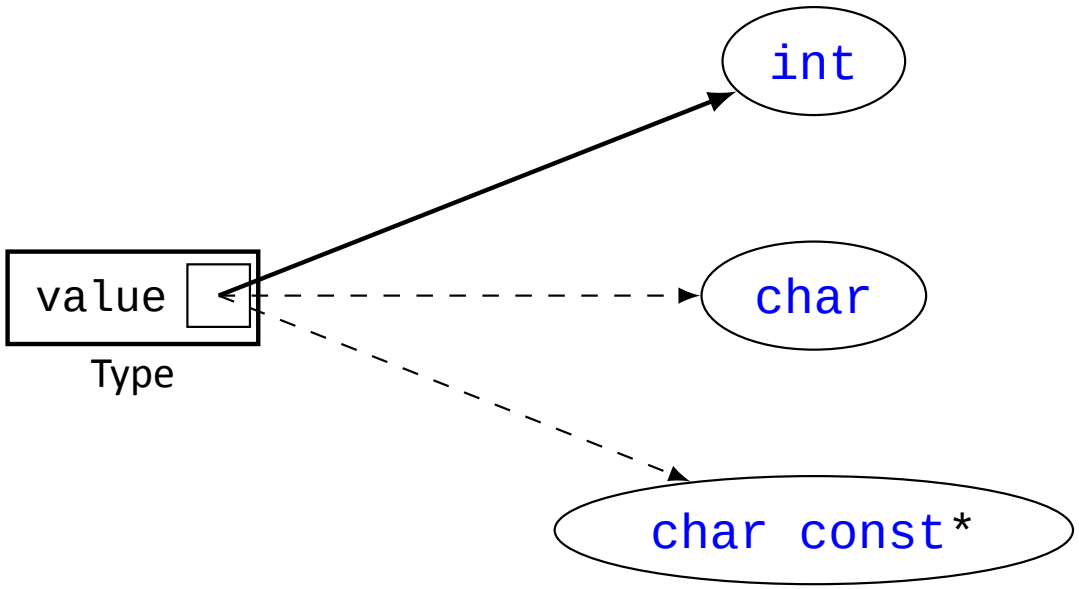
Intro

Sum Type



Intro

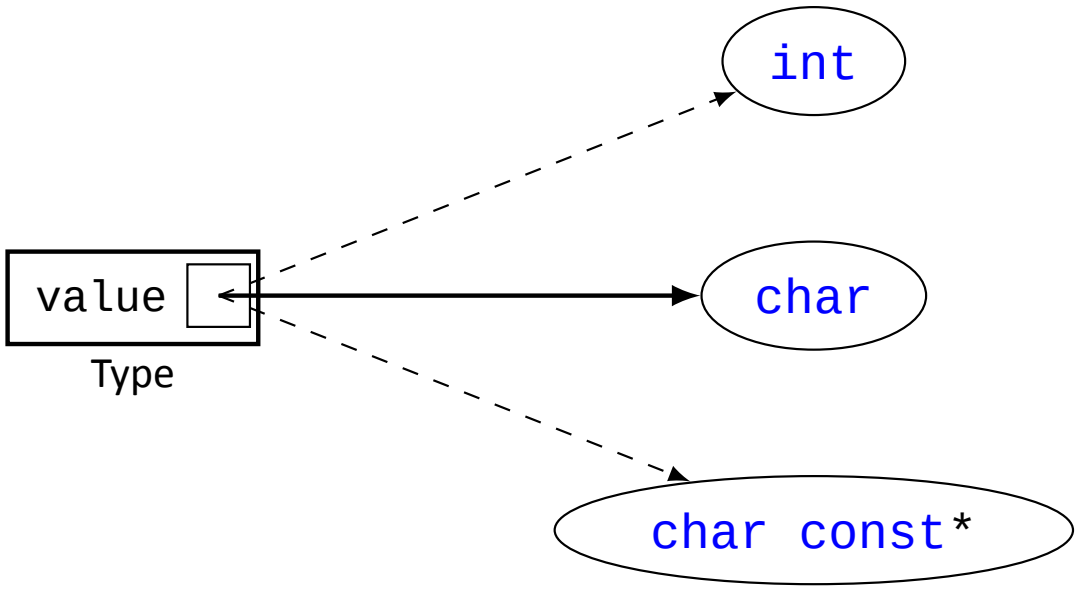
Sum Type




```
value : 5
```

Intro

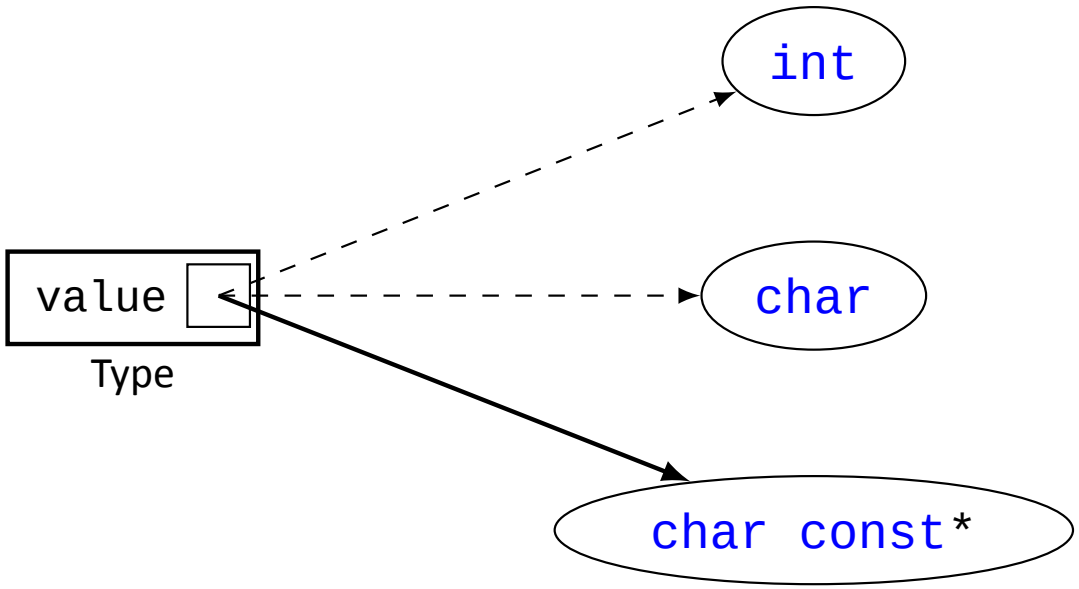
Sum Type




```
value : 'a'
```

Intro

Sum Type




```
value : "some text"
```

Intro

This sounds like Python (sort of)

- use sum types to simulate dynamic types

Intro

This sounds like Python (sort of)

- use sum types to simulate dynamic types
- but how do they work in C++?

LINKÖPING UNIVERSITY

Union

Unions

```
union Sum_Type
{
  int n;
  char c;
  char const* s;
};
```

```
int main()
{
  Sum_Type obj;
  obj.n = 5;
  obj.c = 'a';
  obj.s = "some text";
}
```

Union

Problems with unions

```
int main()
{
  Sum_Type obj;
  obj.n = 5;
  cout << obj.c << endl;
}
```

Union

Problems with unions

```
int main()
{
  Sum_Type obj;
  obj.n = 5;
  cout << obj.c << endl;
}
```

Undefined Behaviour

Union

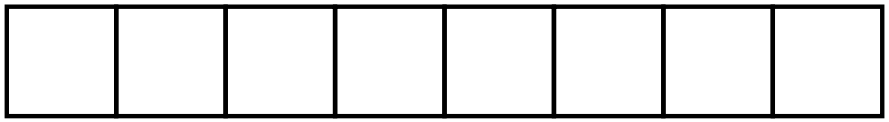
(Possible) Memory model of unions

```
union Sum_Type
{
  int n; // 4 bytes
  char c; // 1 byte
  char const* s; // 8 bytes
};

sizeof(Sum_Type) == 8
```

Union

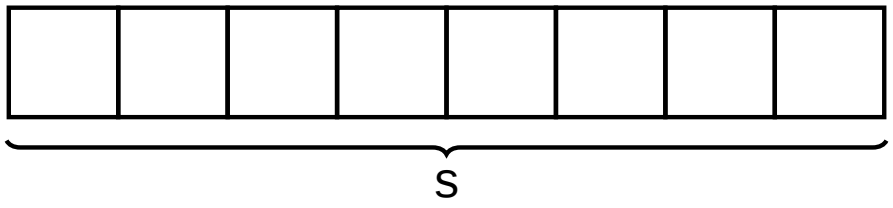
(Possible) Memory model of unions



```
Sum_Type obj;
```

Union

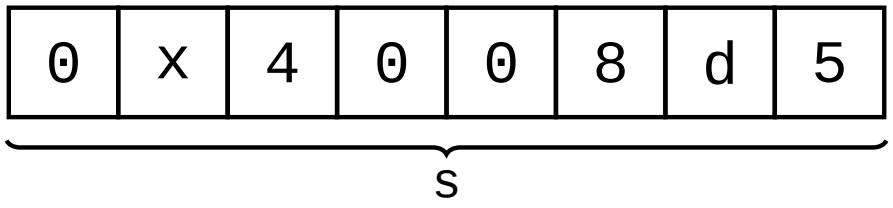
(Possible) Memory model of unions




```
obj.s = "some text";
```

Union

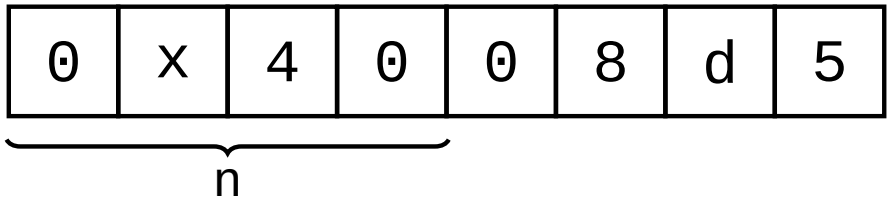
(Possible) Memory model of unions




```
obj.s = "some text";
```

Union

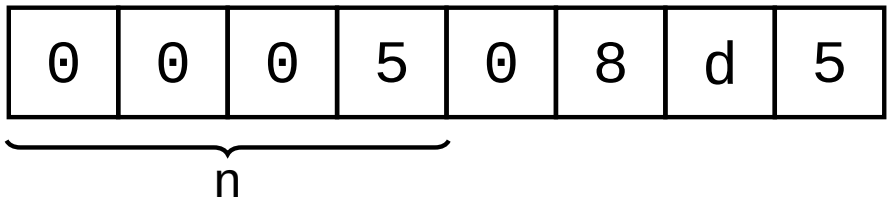
(Possible) Memory model of unions




```
obj.n = 5;
```

Union

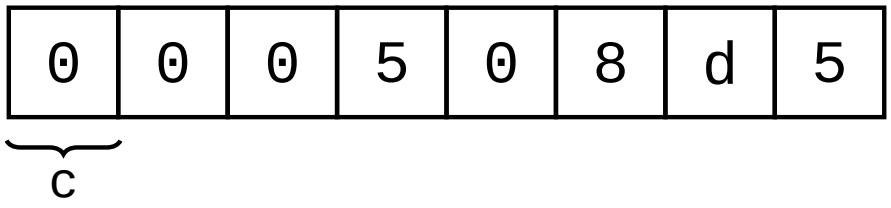
(Possible) Memory model of unions




```
obj.n = 5;
```

Union

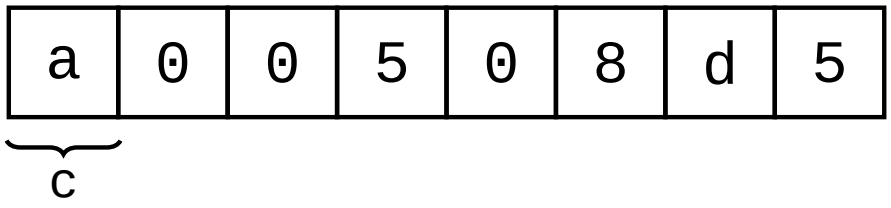
(Possible) Memory model of unions




```
obj.c = 'a';
```

Union

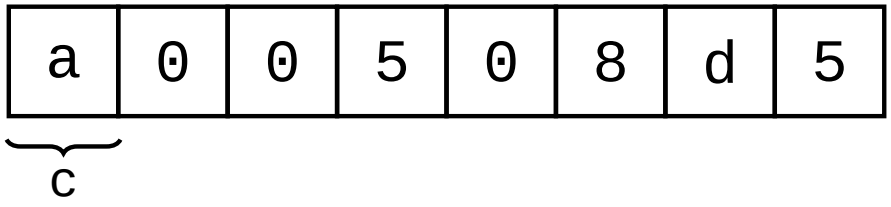
(Possible) Memory model of unions




```
obj.c = 'a';
```

Union

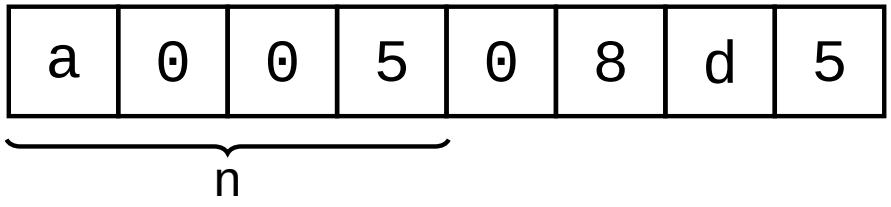
(Possible) Memory model of unions




```
cout << obj.n << endl;
```

Union

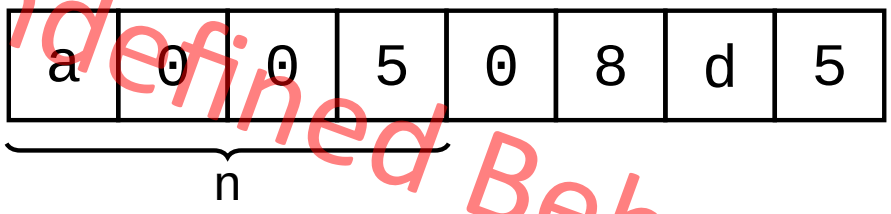
(Possible) Memory model of unions




```
cout << obj.n << endl;
```

Union

(Possible) Memory model of unions



Undefined Behaviour

```
cout << obj.n << endl;
```

LINKÖPING UNIVERSITY

STL types

Sum Types in STL

- `std::optional`
- `std::variant`
- `std::any`

STL types

`std::optional`

```
#include <optional>
// ...
template <typename T>
std::optional<T> read(istream& is)
{
  T data;
  if (is >> data)
  {
    return data;
  }
  return {};
}
```

STL types

std::optional

```
int main()
{
  std::optional<int> result{read<int>(cin)};
  if (result)
  {
    cout << result.value() << endl;
    result = nullopt;
  }
  else
  {
    cout << "Error!" << endl;
  }
}
```

STL types

`std::variant`

```
#include <variant>
// ...
int main()
{
  std::variant<int, double> data{15};

  cout << std::get<int>(data) << endl;

  data = 12.5;

  cout << std::get<1>(data) << endl;
}
```

STL types

std::variant

```
// will initialize data to contain 0
std::variant<int, double> data{};

try
{
  // will throw since data contains int
  cout << std::get<double>(data) << endl;
}
catch (std::bad_variant_access& e) { }

// will assign 12.5 as an int
// so data will contain 12
std::get<int>(data) = 12.5;
```

STL types

std::any

```
#include <any>
// ...
int main()
{
  std::any var;

  var = 5; // int
  cout << std::any_cast<int>(var) << endl;

  var = new double{5.3}; // double*
  cout << *std::any_cast<double*>(var) << endl;
  delete std::any_cast<double*>(var);
}
```

STL types

`std::any`

```
std::any var;
if (var.has_value()) { ... }

var = 7;

if (var.type() == typeid(int)) { ... }

try
{
  cout << std::any_cast<double>(var) << endl;
}
catch (std::bad_any_cast& e) { }
```

LINKÖPING UNIVERSITY

Implementation

Variant

- let us implement a simplified `variant` type

Implementation

Variant

- let us implement a simplified `variant` type
- our variant will store `int` or `std::string`

Implementation

Variant

- let us implement a simplified `variant` type
- our variant will store `int` or `std::string`
- two versions; one with `union` and one without

Implementation

Variant

- let us implement a simplified `variant` type
- our variant will store `int` or `std::string`
- two versions; one with `union` and one without
- we will also introduce a new way to handle memory

Implementation

Union-like classes

```
struct my_union
{
  union
  {
    int n;
    double d;
  };
};
```

```
int main()
{
  my_union m{0};
  cout << m.n << endl;

  m.d = 5.0;
  cout << m.d << endl;
}
```

Implementation

Non-trivial union-like classes

```
struct my_union
{
 union
 {
   int n;
   std::string s;
 };
};
```

```
int main()
{
  my_union u{};

  u.s = "hello";

  cout << u.s << endl;
}
```

Implementation

Non-trivial union-like classes

```
union.cc:14:12: error: use of deleted function 'my_union::my_union()'
   my_union u;
            ^
union.cc:3:8: note: 'my_union::my_union()' is implicitly deleted because
                    the default definition would be ill-formed:
 struct my_union
        ^~~~~~~~
union.cc:14:12: error: use of deleted function 'my_union::~my_union()'
   my_union u;
            ^
union.cc:3:8: note: 'my_union::~my_union()' is implicitly deleted because
                    the default definition would be ill-formed:
 struct my_union
        ^~~~~~~~
```

Implementation

Non-trivial union-like classes

```
struct my_union
{
 my_union() : n{0} { }
 ~my_union() { }
 union
 {
   int n;
   std::string s;
 };
};
```

```
int main()
{
  my_union u{};

  u.s = "hello";

  cout << u.s << endl;
}
```

Implementation

Non-trivial union-like classes

```
struct my_union
{
 my_union() : n{0} { }
 ~my_union() { }
 union
 {
   int n;
   std::string s;
 };
};
```

```
int main()
{
  my_union u{};

  u.s = "hello";

  cout << u.s << endl;
}
```

Segmentation Fault

Implementation

Non-trivial union-like classes

```
struct my_union
{
 my_union() : n{0} { }
 ~my_union() { }
 union
 {
   int n;
   std::string s;
 };
};
```

```
int main()
{
  my_union u{};

  u.s = "hello";

  cout << u.s << endl;
}
```

Why though?!

Implementation

Placement new

```
struct my_union
{
 my_union() : n{0} { }
 ~my_union() { }
 union
 {
   int n;
   std::string s;
 };
};
```

```
int main()
{
  my_union u{};

  new (&u.s) std::string;
  u.s = "hello";

  cout << u.s << endl;
}
```

Implementation

But what about destruction?

```
int main()
{
  my_union u{};

  // call constructor
  new (&u.s) std::string;
  u.s = "hello";

  cout << u.s << endl;

  // explicitly call destructor
  u.s.std::string::~string();
}
```

Implementation

OK, but how do I get correct destruction automatically?

```
struct my_union
{
 my_union() : n{0}, tag{INT} { }
 ~my_union() { }
 union
 {
   int n;
   std::string s;
 };
 enum class Type { INT, STRING };
 Type tag;
};
```

Implementation

Now we are ready for our own implementation

```
class Variant
{
public:
  // ...
private:
  enum class Type { INT, STRING };
  Type tag;
  union
  {
    int n;
    string s;
  };
};
```

Implementation

Now we are ready for our own implementation

```
class Variant
{
public:
  Variant(int n = 0);
  Variant(string const& s);
  ~Variant();
  Variant& operator=(int other) &;
  Variant& operator=(string const& other) &;

  int& num();
  string& str();
  // ...
};
```

Implementation

Constructors

```
Variant::Variant(int n)
  : n{n}, tag{Type::INT}
{ }

Variant::Variant(string const& s)
  : s{s}, tag{Type::STRING}
{ }
```

Implementation

Destructor

```
Variant::~Variant()
{
  if (tag == Type::STRING)
  {
    s.~string();
  }
}
```

Implementation

Assignment operators

```
Variant& Variant::operator=(int other) &
{
  if (tag == Type::STRING)
  {
    s.~string();
  }
  n = other;
  tag = Type::INT;
  return *this;
}
```

Implementation

Assignment operators

```
Variant& Variant::operator=(string const& other) &
{
  if (tag == Type::INT)
  {
    new (&s) string;
  }
  s = other;
  tag = Type::STRING;
  return *this;
}
```

Implementation

Getters

```
int& Variant::num()
{
  if (tag == Type::INT)
  {
    return n;
  }
  throw /* ... */;
}
```

Implementation

Getters

```
string& Variant::str()
{
  if (tag == Type::STRING)
  {
    return s;
  }
  throw /* ... */;
}
```

Implementation

Test program

```
Variant v{}; // will set n = 0
cout << v.num() << endl;

// active field is int
v = 5;
cout << v.num() << endl;

// active field is int, we must
// construct a string inside the variant
v = "this is a long string";
cout << v.str() << endl;

// the destructor must destroy the string here
```

LINKÖPING UNIVERSITY

Second Implementation

Placement new

```
std::string s{};
char data[sizeof(std::string)];
union { int n; std::string s; } u;
int array[sizeof(std::string) / sizeof(int)];
int i{};

new (&s)    std::string; // OK
new (data)  std::string; // OK
new (&u.s)  std::string; // OK
new (array) std::string; // NOT OK
new (&i)    std::string; // NOT OK
```

Second Implementation

Placement new in C-arrays

```
char data[sizeof(std::string)];
std::string* p {new (data) std::string};
*p = "hello world";
p->~string();
```

Second Implementation

Second version (no `union`)

```
class Variant
{
public:
  // ...
private:
  enum class Type { INT, STRING };
  char data[sizeof(string)];
  Type tag;
};
```

Second Implementation

Second version (no union)

```
class Variant
{
public:
  Variant(int n = 0);
  Variant(string const& s);
  ~Variant();
  Variant& operator=(int other) &;
  Variant& operator=(string const& other) &;

  int& num();
  string& str();
  // ...
};
```

Second Implementation

Constructors

```
Variant::Variant(int n)
  : data{}, tag{Type::INT}
{
  new (data) int{n};
}
```

Second Implementation

Constructors

```
Variant::Variant(string const& s)
  : data{}, tag{Type::STRING}
{
  new (data) string{s};
}
```

Second Implementation

Now, how do we retrieve our objects from the array?

```
*reinterpret_cast<string*>(&data)
```

Second Implementation

Now, how do we retrieve our objects from the array?

```
*reinterpret_cast<string*>(&data)
```

Undefined Behaviour

Second Implementation

Aliasing

```
int x{};

// aliases to x
int* p{&x};
int& r{x};

// modifying x through aliases
*p = 5; // OK
r = 7;  // OK
```

Second Implementation

Aliasing

```
int x{};

float* p{reinterpret_cast<float*>(&x)};

*p = 3.7; // NOT OK
```

Second Implementation

Strict aliasing rule

An object of type `T` can be aliased if the alias has one of the following types;

- `T*`
- `T&`
- `char*`
- (`unsigned char*` and `std::byte*`)

Second Implementation

The fix

```
*std::launder(reinterpret_cast<string*>(&data));
```

Second Implementation

Getters

```
int& Variant::num()
{
  if (tag == Type::INT)
  {
    return *std::launder(
              reinterpret_cast<int*>(&data));
  }
  throw /* ... */;
}
```

Second Implementation

Getters

```
string& Variant::str()
{
  if (tag == Type::STRING)
  {
    return *std::launder(
              reinterpret_cast<string*>(&data));
  }
  throw /* ... */;
}
```

Second Implementation

Destructor

```
Variant::~Variant()
{
  if (tag == Type::STRING)
  {
    str().~string();
  }
}
```

Second Implementation

Assignment operators

```
Variant& Variant::operator=(int other) &
{
  if (tag == Type::STRING)
  {
    str().~string();
  }
  tag = Type::INT;
  num() = other;
  return *this;
}
```

Second Implementation

Assignment operators

```
Variant& Variant::operator=(string const& other) &
{
  if (tag == Type::INT)
  {
    new (data) std::string;
  }
  tag = Type::STRING;
  str() = other;
  return *this;
}
```

www.liu.se

LINKÖPING
UNIVERSITY