

TDDD38/726G82 - Advanced programming in C++

Sum Types in C++

Christoffer Holm

Department of Computer and information science

- 1 Intro
- 2 Union
- 3 STL types
- 4 Implementation
- 5 Second Implementation

- 1 **Intro**
- 2 Union
- 3 STL types
- 4 Implementation
- 5 Second Implementation

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
 - A type containing several other types at once
 - `struct` and `class` types are product types
 - `std::pair` and `std::tuple`

Intro

Type categories

Algebraic Data Types

- Product types
- Sum types

Intro

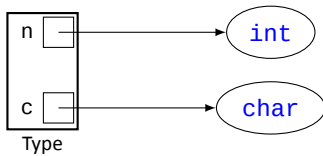
Type categories

Algebraic Data Types

- Product types
- Sum types
 - A sum type is a type that can take on one of several types at a time
 - I.e. a type which can only store one value, but that value might be chosen from more than one type

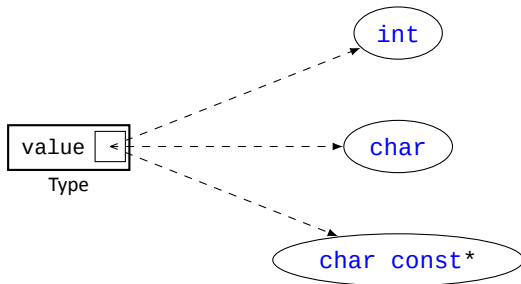
Intro

Product Type



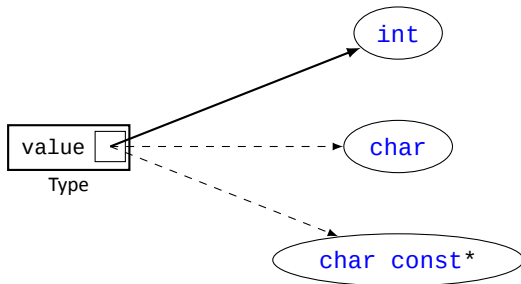
Intro

Sum Type



Intro

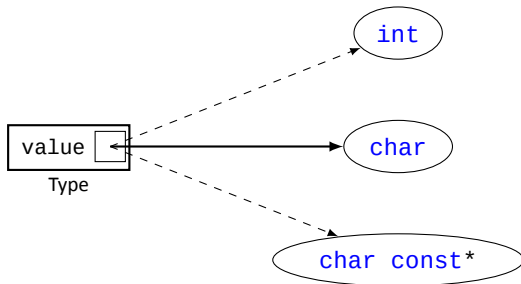
Sum Type



value : 5

Intro

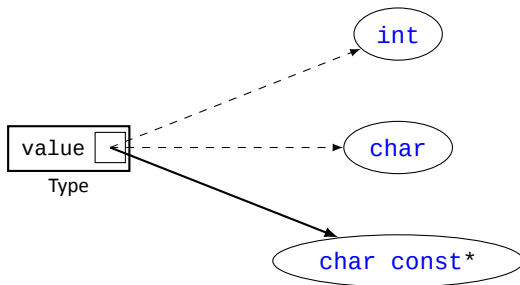
Sum Type



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++?

- 1 Intro
- 2 **Union**
- 3 STL types
- 4 Implementation
- 5 Second Implementation

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

Unions

- Unions look like `struct` or `class`
- They work very different though
- Only one field can be set at one point

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

Problems with unions

- The only field that is safe to access is the one last set
- Accessing any other field will be undefined behaviour
- Once we assign to a new field the old one will be overwritten

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

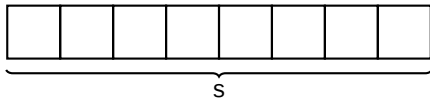
(Possible) Memory model of unions



```
Sum_Type obj;
```


Union

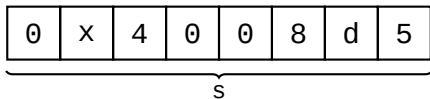
(Possible) Memory model of unions



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

Union

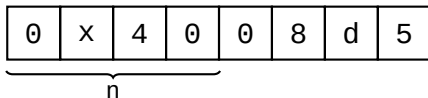
(Possible) Memory model of unions



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

Union

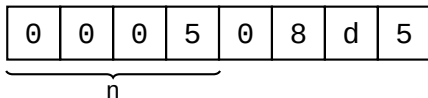
(Possible) Memory model of unions



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

Union

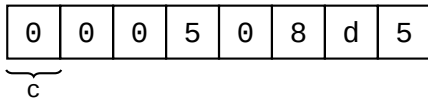
(Possible) Memory model of unions



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

Union

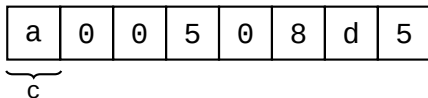
(Possible) Memory model of unions



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

Union

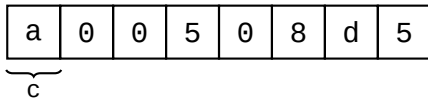
(Possible) Memory model of unions



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

Union

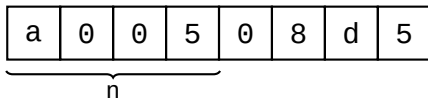
(Possible) Memory model of unions



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

Union

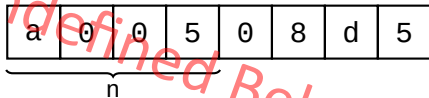
(Possible) Memory model of unions



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


Union

(Possible) Memory model of unions



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

Union

(Possible) Memory model of unions

- All fields in unions are stored in the same memory
- The size of the union is (at least) the size of the largest field; to make sure that everything fits
- Accessing any field other than the one latest assigned to is undefined behaviour
- The memory model presented here is just a common implementation, there are no implementation specifications in the standard

- 1 Intro
- 2 Union
- 3 STL types**
- 4 Implementation
- 5 Second Implementation

STL types

Sum Types in STL

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

STL types

Sum Types in STL

- `std::optional`
 - either stores a value or nothing
 - can only store values of one type with an additional value called `nullopt`
 - often used as a return type so that errors can be reported as `nullopt`
 - will store the value inline inside the object
- `std::variant`
- `std::any`

STL types

Sum Types in STL

- `std::optional`
- `std::variant`
 - a safe alternative to unions
 - will always hold a value of one of several types
 - only possible to access the value as the type it is
 - will throw exceptions when used incorrectly
- `std::any`

STL types

Sum Types in STL

- `std::optional`
- `std::variant`
- `std::any`
 - contains a value of *any* type
 - is extremely general
 - but is very expensive
 - the value is stored on the heap and is polymorphic
 - so `std::any` should be avoided if at all possible

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::variant`

- possible to assign a value to the variant with `operator=`
- use `std::get` to access the value as the correct type
- the `std::variant` will keep track of the current value and type
- throws an `std::bad_variant_access` whenever the user tries to access the incorrect type

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) { }
```

STL types

`std::any`

- `std::any` allows us to store whatever we want
- uses dynamic allocations and `typeid` to keep track of data and type
- is quite inefficient and not that useful
- prefer `std::variant` instead whenever possible

- 1 Intro
- 2 Union
- 3 STL types
- 4 Implementation**
- 5 Second Implementation

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

Union-like classes

- it is possible to create multiple variables that occupy the same memory
- this is done through the use of a so called *anonymous union*
- an anonymous union will create each field inside the class as if they were members, but they will share the same memory space

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

- the compiler is unable to generate constructors and destructors for unions
- this is because the compiler is unable to determine if a fields destructor and constructor should be called
- since only one type can be active at once the compiler can't know which one it is (if any)
- due to this, we must define special member functions ourselves

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;
}
```

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

Non-trivial union-like classes

- only one field is active at once
- in the constructor we initialize `n`
- thus leaving `s` uninitialized
- when we assign to `s` we are assigning to an uninitialized string

Implementation

Non-trivial union-like classes

- assignment assumes that both strings are correctly initialized
- we would have to call a constructor on s
- ... but this can only be done at initialization?
- there is one other way to call constructors after the fact!

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

Placement `new`

- placement `new` is a call to `new` with an extra parameter
- this extra parameter is a pointer to memory where an object should be placed
- this will **not** allocate any memory
- but will instead call a constructor of a specified type on the specified memory location
- this is a way to manually handle lifetime without any dynamic allocations!

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

But what about destruction?

- unions does not track which field is active
- so the compiler will be unable to call the appropriate destructor
- the `my_union` destructor is unable to know which field is active
- therefore we have to manually call the destructor of `s` to ensure that no memory leaks occur
- calling the `string` destructor will only work if the union actually contains a `string`

Implementation

Extra note

- `u.s.std::string::~~string()` is the way we call the destructor
- if we have `using std::string` or `using namespace std` in our code we can simplify this to `u.s.~string()`
- `std::string` is in reality an alias for `std::basic_string<char>` so we can also write `u.s.~basic_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

OK, but how do I get correct destruction automatically?

- the only way to correctly destroy objects is if we ourselves keep track of what the current type is
- we create a so called *tagged union*
- we have some kind of data member that tracks what the current type is stored
- we will of course have to update this tag whenever we change the type

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

Union-based implementation

- we create our variant as a tagged union
- use the tag data member to keep track of which type is currently stored
- we have assignment and getters as our interface
- will have to always check the type before performing operations

Implementation

Constructors

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

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

Implementation

Constructors

- the constructors will initialize the appropriate field in the union
- they will also initialize tag to the appropriate value

Implementation

Destructor

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

Implementation

Destructor

- if the currently assigned value is of type `int` then nothing needs to be done
- however; if the active type is `string` we have to manually call the destructor on that field

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

Assignment operators

- if we are assigning a string we must guarantee that `s` is an initialized string object
- if the active field is not `string` in that case we have to use placement new to construct a string in `s`
- if we are assigning an `int` we must potentially destroy `s` (if `s` was the previous active field)
- therefore we check the type and call the destructor if necessary

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

Getters

- the getters should only return valid values
- therefore we throw some kind of exception if the active field is of incorrect type

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
```

- 1 Intro
- 2 Union
- 3 STL types
- 4 Implementation
- 5 **Second Implementation**

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

- We can place our object in any memory that is;
- a `union`
- a `char` array with enough space
- or an object of the same type as the one we are trying to construct

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)
```

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

Strict aliasing rule

accessing objects through pointers or references is known as *aliasing*.

- so when aliasing an object of type T the following must be true;
- must be accessed through a T pointer or reference
- or must be accessed through a **char** pointer
- otherwise this is undefined behaviour

This is known as the *strict aliasing rule*

Second Implementation

The fix

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

Second Implementation

`std::launder`

- `std::launder` is defined in `<new>`
- tell the compiler that it must ignore the strict aliasing rule in this case
- **Note:** only correct if we are trying to point to an actually constructed object of the specified type

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