TDDD38/726G82:
Adv. Programming in C++
Advanced Memory II

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1. Unions
2. General-purpose storage
3. Non-global allocation (Bonus material)

1. Unions
2. General-purpose storage
3. Non-global allocation (Bonus material)

Unions

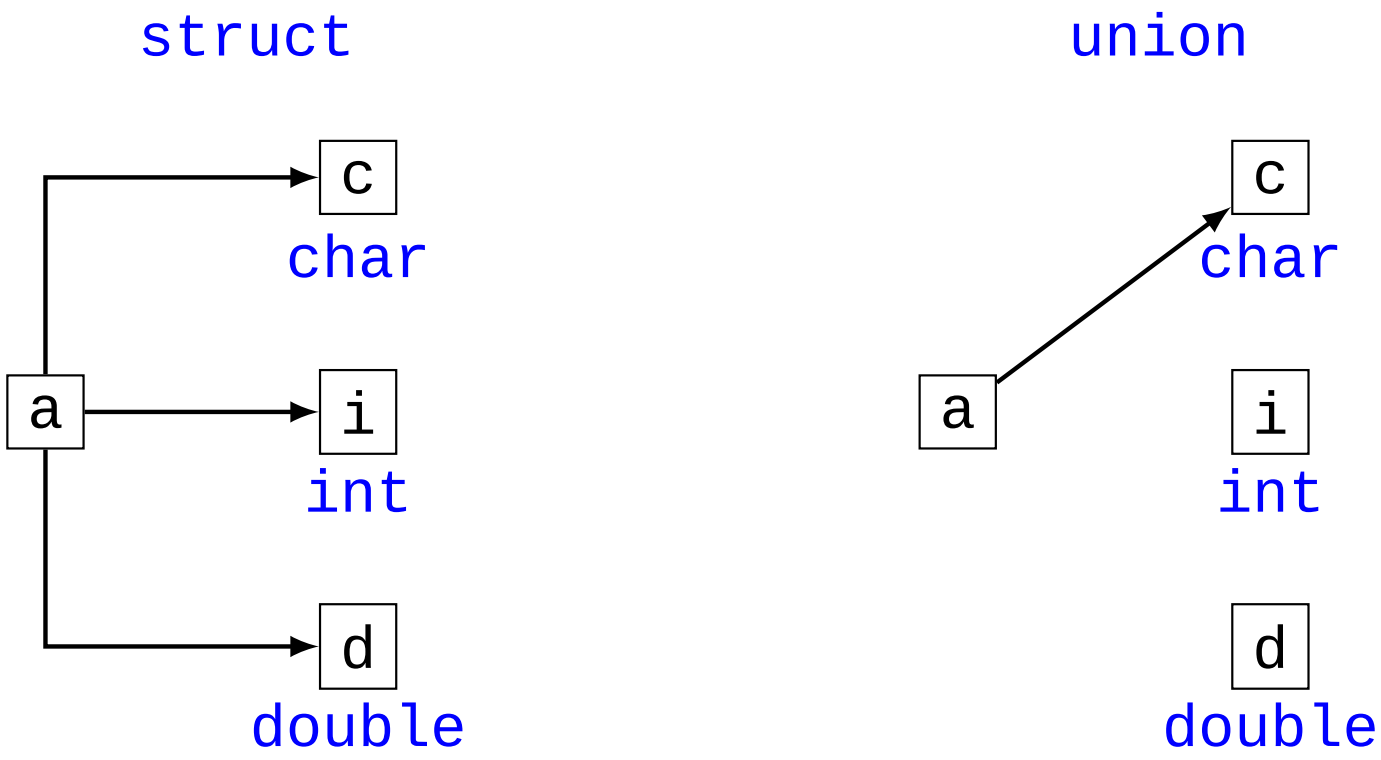
What are unions?

```
1  struct My_Type
2  {
3    char c;
4    int i;
5    double d;
6  };
7
8  My_Type obj { 'a' };
```

```
1  union My_Type
2  {
3    char c;
4    int i;
5    double d;
6  };
7
8  My_Type obj { 'a' };
```

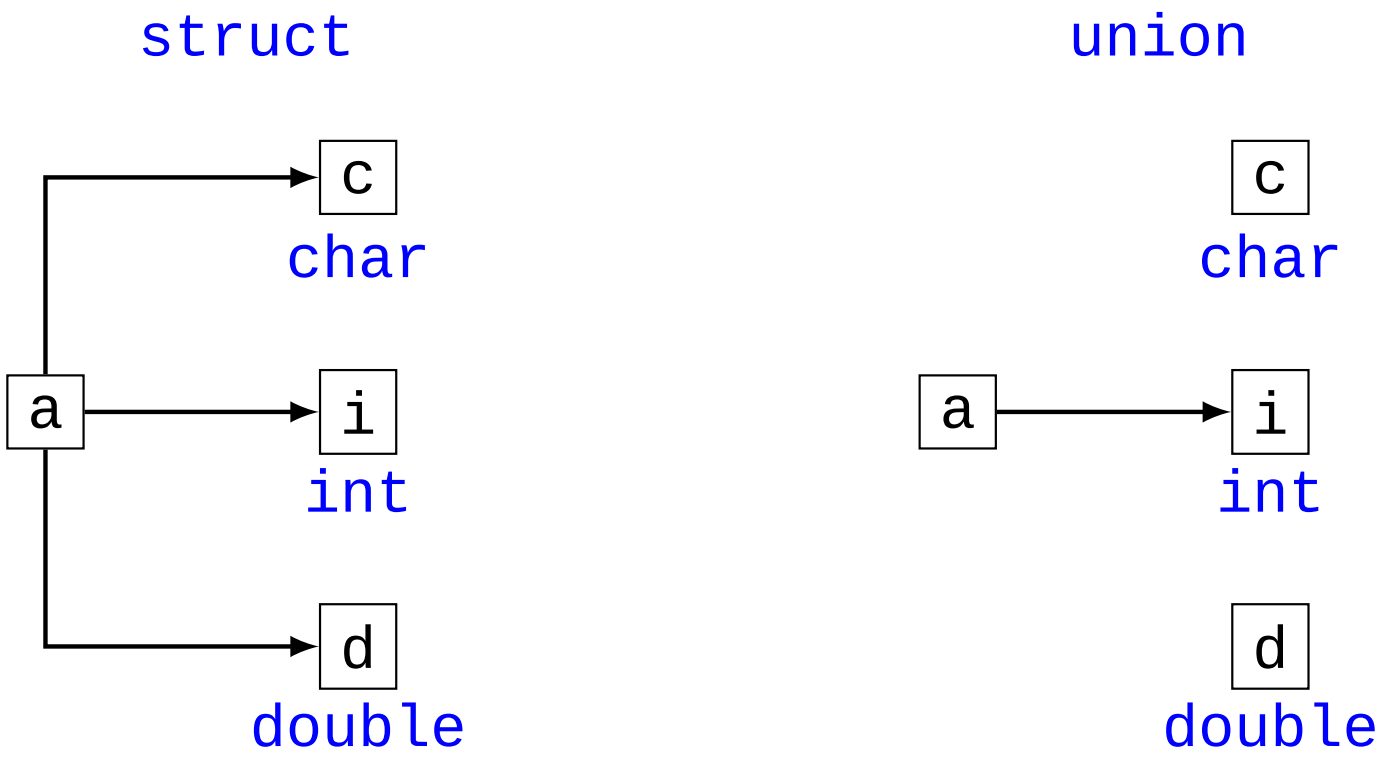
Unions

What are unions?



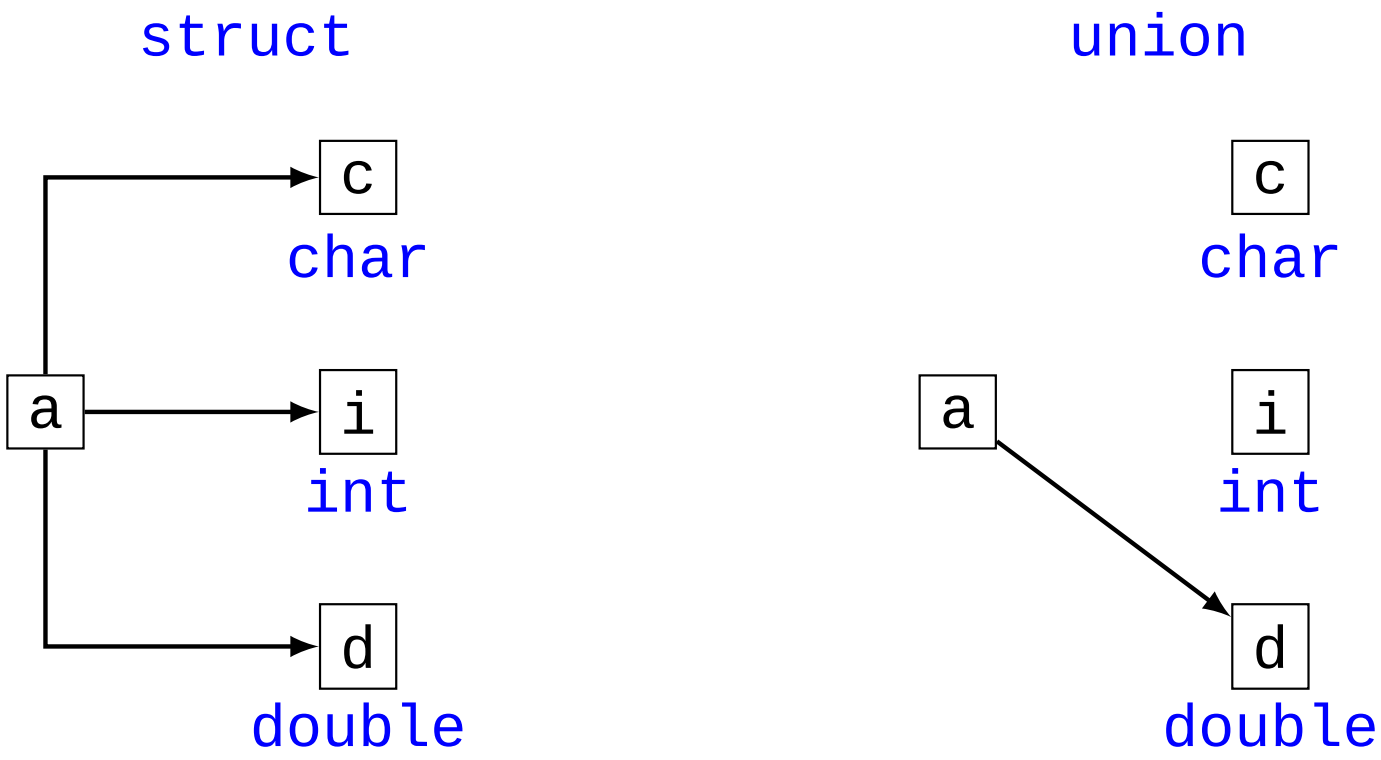
Unions

What are unions?



Unions

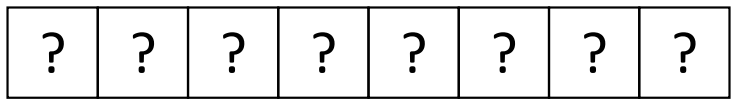
What are unions?



Unions

Unions: memory model

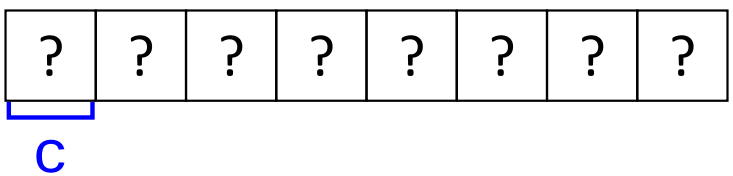
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

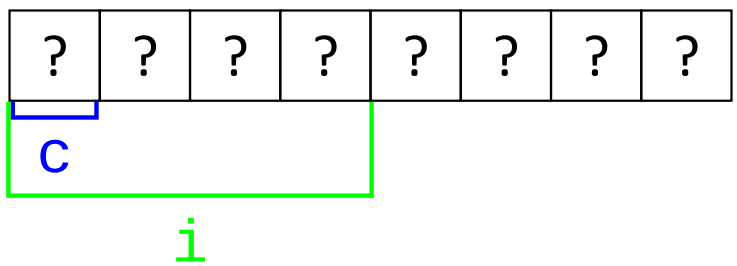
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

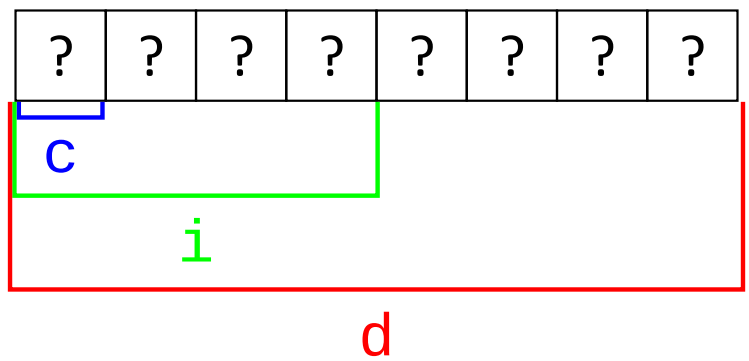
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

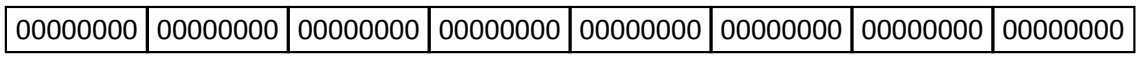
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Using unions

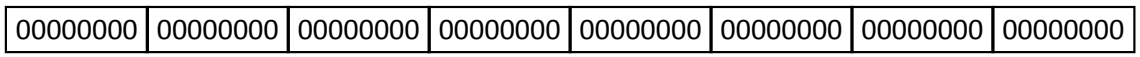
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```

a: 

Unions

Using unions

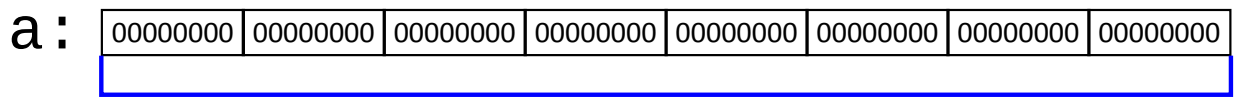
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```

a: 

Unions

Using unions

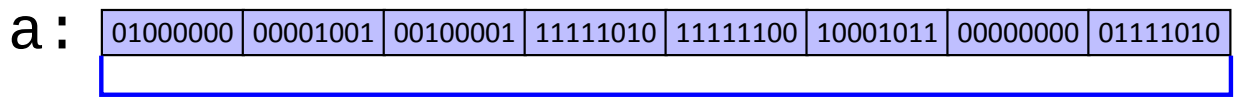
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```

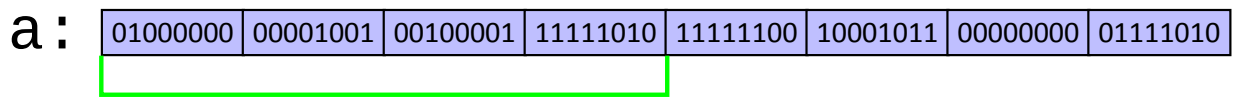
a:

01000000	00001001	00100001	11111010	11111100	10001011	00000000	01111010

Unions

Using unions

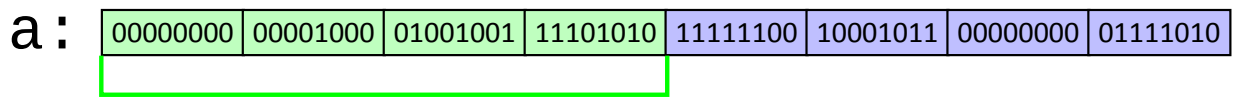
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

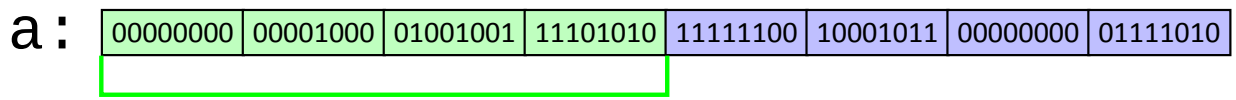
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

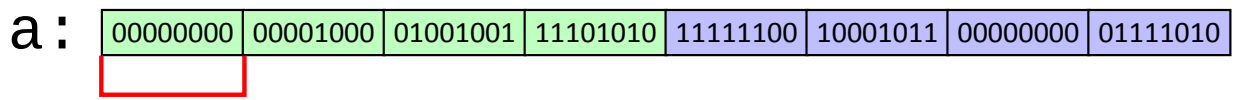
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

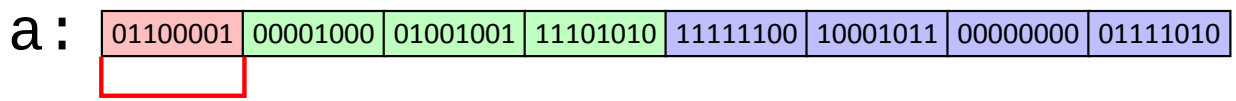
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

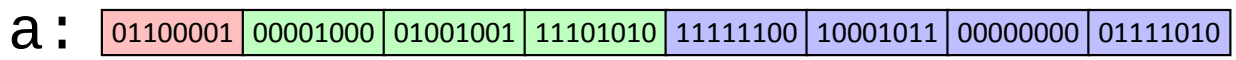
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Non-trivial unions

```
1  union My_Union
2  {
3    double value;
4    std::string text;
5  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

Unions

Error!

```
union.cc: In function 'int main()':
union.cc:14:31: error: use of deleted function 'My_Union::~My_Union()'
   14 |     My_Union u { .value = 1.0 };
      |                               ^
union.cc:6:7: note: 'My_Union::~My_Union()' is implicitly deleted because
the default definition would be ill-formed:
    6 | union My_Union
      |       ^~~~~~~~
union.cc:9:15: error: union member 'My_Union::text' with non-trivial
'std::__cxx11::basic_string<_CharT, _Traits, _Alloc>::~basic_string()
[with _CharT = char; _Traits = std::char_traits<char>; _Alloc =
std::allocator<char>]'
    9 |   std::string text;
      |               ^~~~
```

Unions

Error!

```
union.cc: In function 'int main()':
union.cc:14:31: error: use of deleted function 'My_Union::~My_Union()'
   14 |     My_Union u { .value = 1.0 };
      |                               ^
union.cc:6:7: note: 'My_Union::~My_Union()' is implicitly deleted because
the default definition would be ill-formed:
    6 | union My_Union
      |       ^~~~~~~~
union.cc:9:15: error: union member 'My_Union::text' with non-trivial
'std::string::~string()'


    9 |   std::string text;
      |               ^~~~
```

Unions

Force a trivial destructor

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

Unions

Force a trivial destructor

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

Segmentation fault!

Unions

Non-trivial members

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

Unions

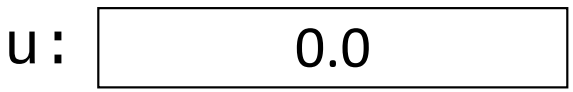
Non-trivial members

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

Unions

Non-trivial members

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```



Unions

Non-trivial members

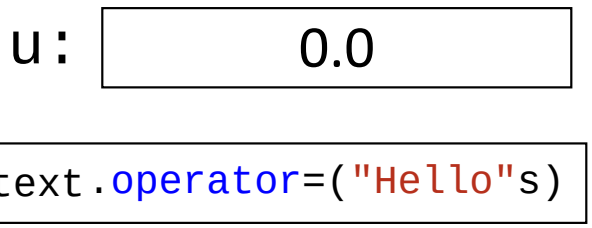
```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

u: 0.0

Unions

Non-trivial members

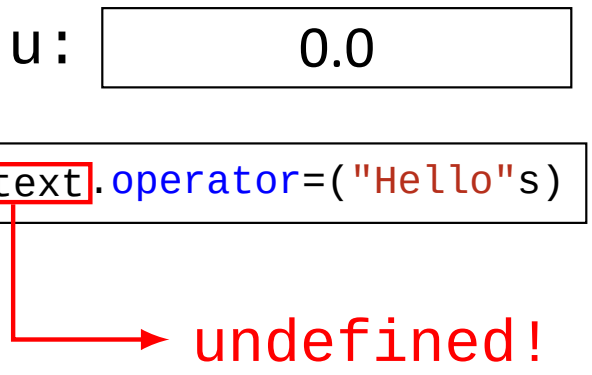
```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```



Unions

Non-trivial members

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```



Unions

Unsuccessful solution

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .text = "" };
4
5    u.text = "Hello"s; // works!
6    cout << u.text << endl;
7
8    u.value = 3.14;
9    cout << u.value << endl;
10
11   u.text = "Bye"s; // ?
12 }
```

Unions

Unsuccessful solution

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .text = "" };
4
5    u.text = "Hello"s; // works!
6    cout << u.text << endl;
7
8    u.value = 3.14;
9    cout << u.value << endl;
10
11   u.text = "Bye"s; // ?
12 }
```

Unions

Unsuccessful solution

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .text = "" };
4
5    u.text = "Hello"s; // works!
6    cout << u.text << endl;
7
8    u.value = 3.14;
9    cout << u.value << endl;
10
11   u.text = "Bye"s; // ?
12 }
```

Works!

Unions

Unsuccessful solution

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .text = "" };
4
5    u.text = "Hello"s; // works!
6    cout << u.text << endl;
7
8    u.value = 3.14;
9    cout << u.value << endl;
10
11   u.text = "Bye"s; // ?
12 }
```

Unions

Unsuccessful solution

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .text = "" };
4
5    u.text = "Hello"s; // works!
6    cout << u.text << endl;
7
8    u.value = 3.14;
9    cout << u.value << endl;
10
11   u.text = "Bye"s; // ?
12 }
```

Segmentation fault...

Unions

How can we call the string constructor?

Unions

How can we call the string constructor?
`operator new`( ) without allocation?

Unions

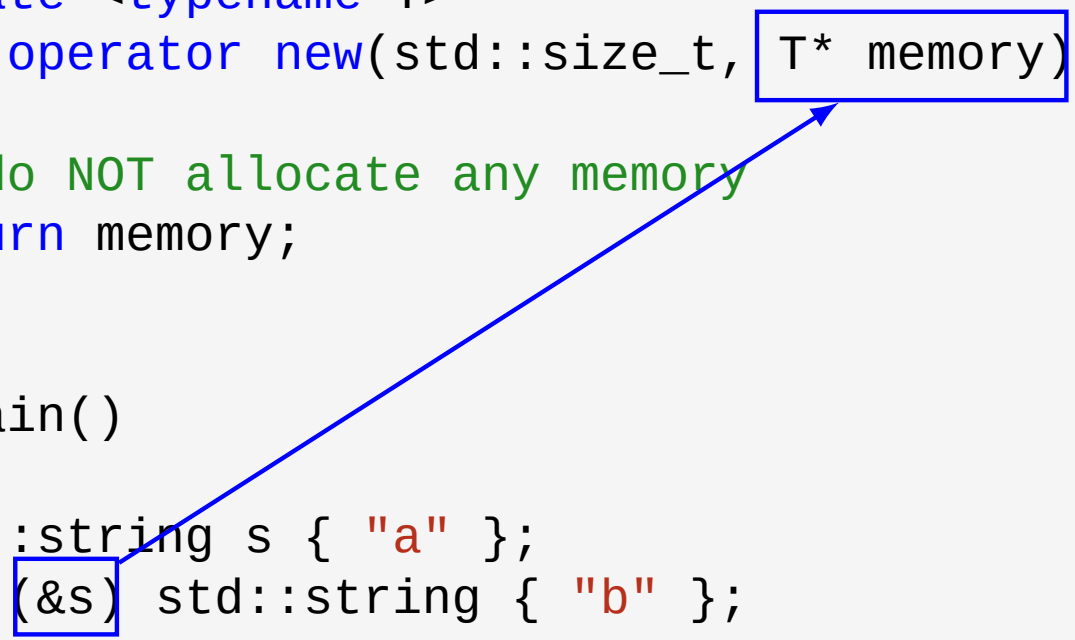
`operator new( )` with custom parameters

```
1  template <typename T>
2  void* operator new(std::size_t, T* memory)
3  {
4    // do NOT allocate any memory
5    return memory;
6  }
7
8  int main()
9  {
10   std::string s { "a" };
11   new (&s) std::string { "b" };
12 }
```

Unions

`operator new( )` with custom parameters

```
1  template <typename T>
2  void* operator new(std::size_t, T* memory)
3  {
4    // do NOT allocate any memory
5    return memory;
6  }
7
8  int main()
9  {
10   std::string s { "a" };
11   new (&s) std::string { "b" };
12 }
```



Unions

`operator new( )` with custom parameters

```
1  #include <new>
2
3  int main()
4  {
5    std::string s { "a" };
6
7    // called 'placement' new
8    new (&s) std::string { "b" };
9  }
```

Unions

Solution to our problems

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    new (&u.text) std::string {};
7    u.text = "Hello"s;
8    cout << u.text << endl;
9
10
11   u.value = 3.14;
12   cout << u.value << endl;
13 }
```

Unions

Solution to our problems

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

Works!

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    new (&u.text) std::string {};
7    u.text = "Hello"s;
8    cout << u.text << endl;
9
10
11   u.value = 3.14;
12   cout << u.value << endl;
13 }
```

Unions

Solution to our problems

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    new (&u.text) std::string {};
7    u.text = "Hello"s;
8    cout << u.text << endl;
9
10
11   u.value = 3.14;
12   cout << u.value << endl;
13 }
```

But leaks memory...

Unions

Fixing the memory leaks

```
1  union My_Union
2  {
3    ~My_Union() { }
4
5    double value;
6    std::string text;
7  };
```

```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    new (&u.text) std::string {};
7    u.text = "Hello"s;
8    cout << u.text << endl;
9    std::destroy_at(&u.text);
10
11   u.value = 3.14;
12   cout << u.value << endl;
13 }
```

Unions

`std::destroy_at()`

```
1  template <typename T>
2  void destroy_at(T* ptr)
3  {
4    ptr->~T();
5  }
```

Unions

Fixes

1. active member changes to non-trivial ⇒ use *placement new*
2. active member is already the type ⇒ use `operator=()`
3. active member changes from non-trivial ⇒ use `std::destroy_at()`

Unions

Fixes

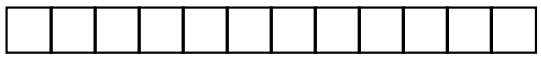
1. active member changes to non-trivial ⇒ use *placement new*
2. active member is already the type ⇒ use `operator=()`
3. active member changes from non-trivial ⇒ use `std::destroy_at()`

Can we make this easier?

Unions

Shared data members

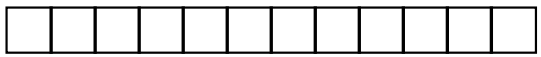
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

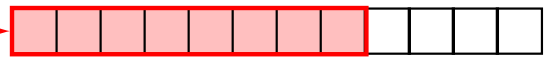
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

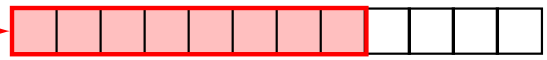
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

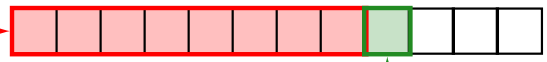
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

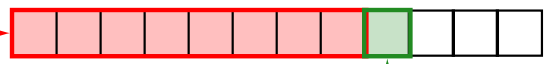
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

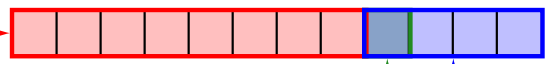
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

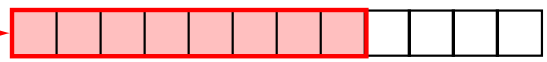
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

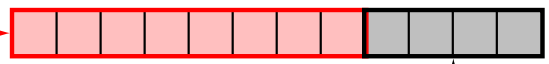
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Tagged union: how to keep track of active members

```
1  enum Type { DOUBLE, STRING };
2  struct My_Union
3  {
4    My_Union();
5    ~My_Union();
6
7    void set_type(Type next);
8
9    // type of the active member
10   Type type;
11
12   // the actual union data members
13   union
14   {
15     double value;
16     std::string text;
17   };
18
19 };
```

```
1  My_Union::My_Union()
2    : type { DOUBLE }, value { 0.0 }
3  {
4  }
5
6  My_Union::~My_Union()
7  {
8    if (type == STRING)
9      std::destroy_at(&text);
10 }
11
12 void My_Union::set_type(Type next)
13 {
14   if (type == next) return;
15   if (next == STRING)
16     new (&text) std::string{};
17   else
18     std::destroy_at(&text);
19 }
```

Unions

Tagged union: Usage

```
1  int main()
2  {
3    My_Union u { };
4
5    u.set_type(DOUBLE);
6    u.value = 3.14;
7
8    u.set_type(STRING);
9    u.text = "Hello";
10
11   // ...
12 }
```

Unions

Anti-pattern: Problematic usage of unions

```
1 // trivial union
2 union Split
3 {
4   double f;
5   int i;
6 };
```

```
1 // interpret float as int
2 int main()
3 {
4   Split s { .f = 3.7 };
5   cout << s.i << endl;
6 }
```

Unions

Anti-pattern: Why it is problematic

source code:

```
1 // init i to 0
2 Split s { .i = 0 };
3
4 // write to f
5 s.f = 3.5;
6
7 // read from i
8 cout << s.i << endl;
```

transformation:

```
1 // i is never changed
2
3 // f is written to but
4 // never read, ignore it
5
6 // print unchanging i
7 cout << 0 << endl;
```

Unions

Strict aliasing

An object of type T can be accessed as:

- T&
- T&&
- T*
- `char*`
- `char const*`

Unions

Strict aliasing

An object of type T can be accessed as:

- T&
- T&&
- T*
- `char*`
- `char const*`
- Anything else counts as a different object

Unions

```
1  class Variant
2  {
3  public:
4    Variant();
5    Variant(double value);
6    Variant(string const& text);
7
8    // ...
9    // five special members
10   // ...
11
12   double& get_value();
13   string& get_string();
14
15   bool has_string() const;
16   bool has_value() const;
17
18   Variant& operator=(double value);
19   Variant& operator=(string const& text);
20 private:
21   My_Union data;
22 };
```

Unions

Generalization of `Variant`

```
1  template <typename... Ts>
2  class Variant
3  {
4  public:
5    // ...
6    template <typename T>
7    T& get();
8
9    template <typename T>
10   bool has() const;
11
12   template <typename T>
13   Variant& operator=(T const& value);
14
15 private:
16   // how to store???
17 };
```

1 Unions
2 General-purpose storage
3 Non-global allocation (Bonus material)

General-purpose storage

Generalization of `Variant`

```
1 template <typename... Ts>
2 class Variant
3 {
4   // ...
5 };
```

Storage:

General-purpose storage

Generalization of `Variant`

```
1 template <typename... Ts>
2 class Variant
3 {
4   // ...
5 };
```

Storage:

1. must be well-defined

General-purpose storage

Generalization of `Variant`

```
1 template <typename... Ts>
2 class Variant
3 {
4   // ...
5 };
```

Storage:

1. must be well-defined
2. must fit largest type in `Ts`

General-purpose storage

Generalization of `Variant`

```
1 template <typename... Ts>
2 class Variant
3 {
4   // ...
5 };
```

Storage:

1. must be well-defined
2. must fit largest type in `Ts`
3. must be possible to read

General-purpose storage

Generalization of `Variant`

```
1 template <typename... Ts>
2 class Variant
3 {
4   // ...
5 };
```

Storage:

1. must be well-defined
2. must fit largest type in `Ts`
3. must be possible to read

We handle these one at a time!

General-purpose storage

How to store *anything*

- Strict aliasing

General-purpose storage

How to store *anything*

- Strict aliasing
- $\Rightarrow$ T readable as `char *`

General-purpose storage

How to store *anything*

- Strict aliasing
- $\Rightarrow$ T readable as `char *`
- $\Rightarrow$ we can *store* anything in `char` array

General-purpose storage

Storing *anything* in `char`-arrays

```
1  template <typename T>
2  void transfer(T const& object, char* target)
3  {
4   // This does NOT create a new T object
5   auto it = reinterpret_cast<char const*>(&object);
6   std::copy(it, it + sizeof(T), target);
7  }
8  int main()
9  {
10  char memory[sizeof(double)];
11  transfer(3.14, &memory[0]);
12 }
```

General-purpose storage

Even better way!

```
1  template <typename T>
2  T* transfer(T const& object, char* target)
3  {
4    // placement new with copy-constructor
5    // this DOES create a new object
6    T* ptr = new (target) T { object };
7
8    // ptr is completely safe to use,
9    // so we return it to the user
10   return ptr;
11 }
```

General-purpose storage

Finding largest type in Ts

```
1  // primary template, variadic recursion
2  template <typename T, typename... Ts>
3  constexpr std::size_t const max_size {
4    std::max(sizeof(T), max_size<Ts...>)
5  };
6
7  // base case, specialization with Ts = {}
8  template <typename T>
9  constexpr std::size_t const max_size<T> {
10   sizeof(T)
11 };
```

General-purpose storage

Combining these things

```
1  template <typename... Ts>
2  class Variant
3  {
4  public:
5    // ...
6  private:
7    template <typename T>
8    void write(T const& data)
9    {
10     auto ptr = transfer(data, &memory[0]);
11     // what to do with ptr?
12   }
13
14   char memory[max_size<Ts...>];
15 };
```

General-purpose storage

Is this a valid way of accessing the data?

`ptr`

General-purpose storage

Is this a valid way of accessing the data?

General-purpose storage

Is this a valid way of accessing the data?

ptr

But then we must save it...

General-purpose storage

Is this a valid way of accessing the data?

```
reinterpret_cast<T*>(&memory[0])
```

General-purpose storage

Is this a valid way of accessing the data?

```
reinterpret_cast<T*>(&memory[0])
```

Undefined behaviour!

General-purpose storage

Is this a valid way of accessing the data?

```
std::launder(reinterpret_cast<T*>(&memory[0]))
```

General-purpose storage

Putting it together (somewhat)

```
 1 template <typename... Ts>
 2 class Variant {
 3 public:
 4   // ...
 5   template <typename T>
 6   T& get() {
 7     return *std::launder(
 8       reinterpret_cast<T*>(&memory[0])
 9     );
10   }
11 private:
12   char memory[max_size<Ts...>];
13 };
```

1 Unions
2 General-purpose storage
3 Non-global allocation (Bonus material)

Non-global allocation

Inefficient allocation pattern

```
1  struct Particle
2  {
3    double x, y, z;
4  };
```

```
1  // remove dead objects
2  auto it = remove_if(begin(objs), end(objs),
3                      is_alive);
4
5  // deallocate dead particles
6  for_each(it, end(objs), [](auto p) {
7    delete p;
8  });
9
10 // actually remove them
11 objs.erase(it, end(objs));
12
13 // generate new objects
14 while (new_particles > 0)
15 {
16   // randomize x, y and z
17   objs.push_back(new Particle { x, y, z });
18   --new_particles;
19 }
```

Non-global allocation

Reusing allocations if possible

```
1  struct Particle
2  {
3    double x, y, z;
4
5    static void* operator new(size_t size) {
6      if (!free.empty()) {
7        auto result = free.back();
8        free.pop_back();
9        return result;
10     }
11     return std::malloc(size);
12   }
13
14   static void operator delete(void* ptr) {
15     free.push_back(ptr);
16   }
17
18 private:
19   static std::vector<void*> free;
20 };
```

Non-global allocation

Slightly more general version

```
1  struct Particle
2  {
3    double x, y, z;
4
5    static void* operator new(size_t size) {
6      if (!free.empty()) {
7        auto result = free.back();
8        free.pop_back();
9        return result;
10     }
11     return ::operator new(size);
12   }
13
14   static void operator delete(void* ptr) {
15     free.push_back(ptr);
16   }
17
18 private:
19   static std::vector<void*> free;
20 };
```

Non-global allocation

A different scenario

```
1  extern vector<double> results;
2
3  // we assume this function is called many times each second
4  void process(istream& is)
5  {
6    // read values from the input (producing many allocations)
7    vector<double> values {
8      istream_iterator<double>{ is },
9      istream_iterator<double>{ }
10   };
11
12   // remove all negative values
13   auto it = remove_if(begin(values), end(values),
14                       [](double x) { return x < 0; });
15   values.erase(it, end(values));
16
17   // sum the result and store it in the persistent data
18   results.push_back(accumulate(begin(values), end(values), 0.0));
19 }
```

Non-global allocation

Enter: Allocators

```
1 template <
2   typename T,
3   typename Allocator = std::allocator<T>
4 > class vector;
```

Non-global allocation

Enter: Allocators

```
1 template <
2   typename T,
3   typename Allocator = std::allocator<T>
4 > class vector;
```

What is this?

Non-global allocation

Simplified implementation

```
1  template <typename T>
2  struct allocator
3  {
4    using value_type = T;
5    using size_type = size_t;
6    using difference_type = ptrdiff_t;
7
8    static T* allocate(size_t n)
9    {
10     return reinterpret_cast<T*>(::operator new(n));
11   }
12
13   static void deallocate(T* ptr, size_t)
14   {
15     ::operator delete(ptr);
16   }
17 };
```

Non-global allocation

Temporary allocator

```
template <typename T>
struct arena_allocator {
  /* type aliases from before */

  T* allocate(size_t) {
      auto ptr = tail;
      tail += sizeof(T);
      return reinterpret_cast<T*>(ptr);
  }

  void deallocate(T* mem, size_t) {
      auto ptr = reinterpret_cast<char*>(mem);
      if (ptr + sizeof(T) >= tail)
          tail -= sizeof(T);
  }

  char buffer[1024] { };
  char* tail { &buffer[0] };
};
```

Non-global allocation

Putting it together

```
1  extern vector<double> results;
2
3  // we assume this function is called many times each second
4  void process(istream& is)
5  {
6    // read values from the input (using arena_allocator)
7    vector<double, arena_allocator<double>> values {
8      istream_iterator<double>{ is },
9      istream_iterator<double>{ }
10   };
11
12   // remove all negative values
13   auto it = remove_if(begin(values), end(values),
14                       [](double x) { return x < 0; });
15   values.erase(it, end(values));
16
17   // sum the result and store it in the persistent data
18   results.push_back(accumulate(begin(values), end(values), 0.0));
19 }
```

www.liu.se

LINKÖPING UNIVERSITY