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

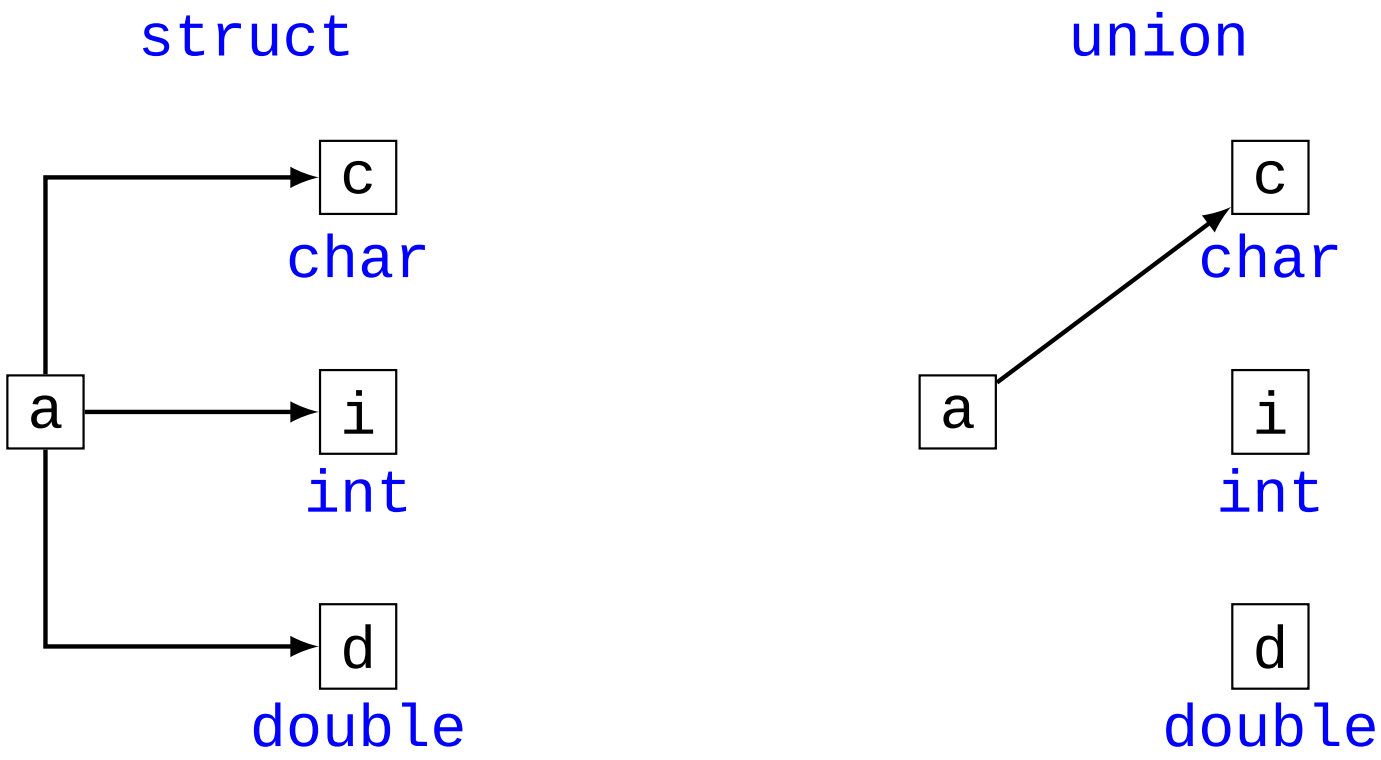
What are unions?

```
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' };
```

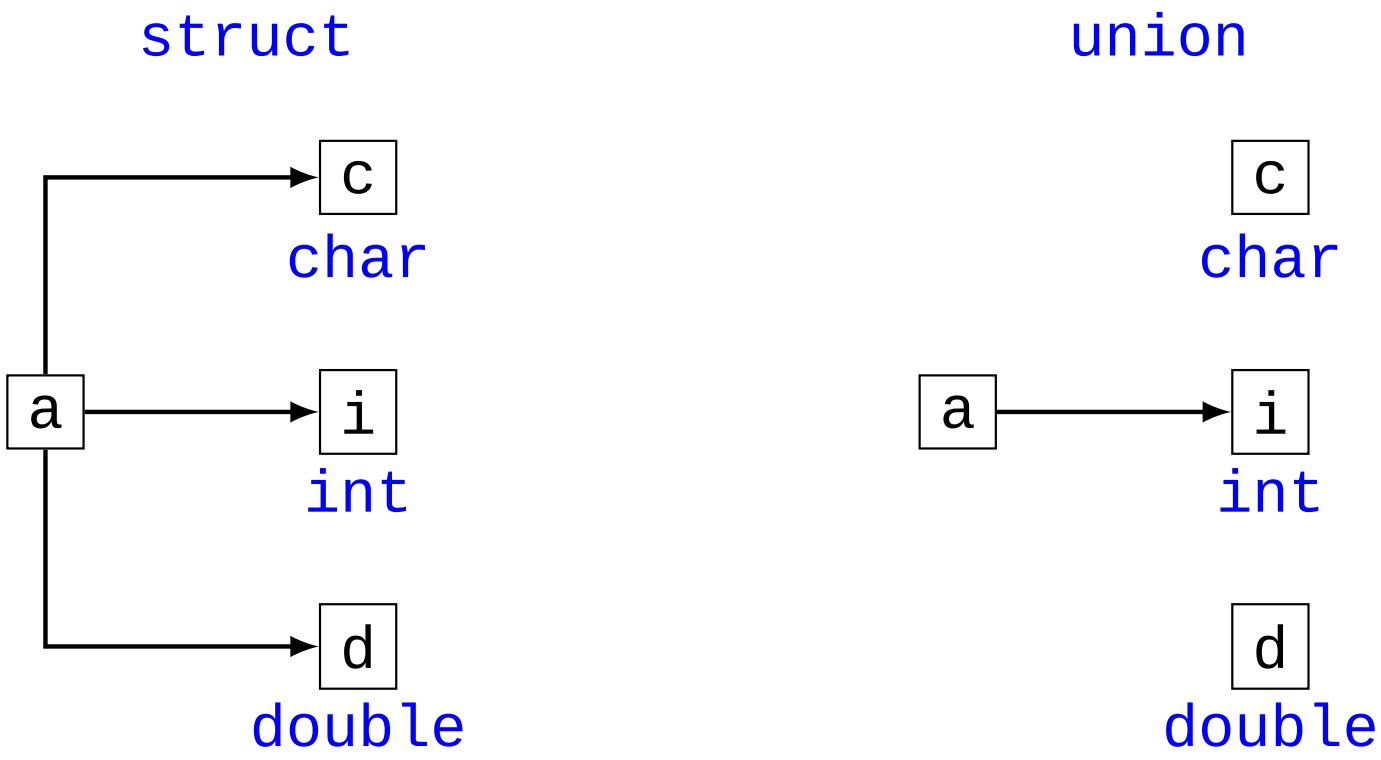
Unions

What are unions?



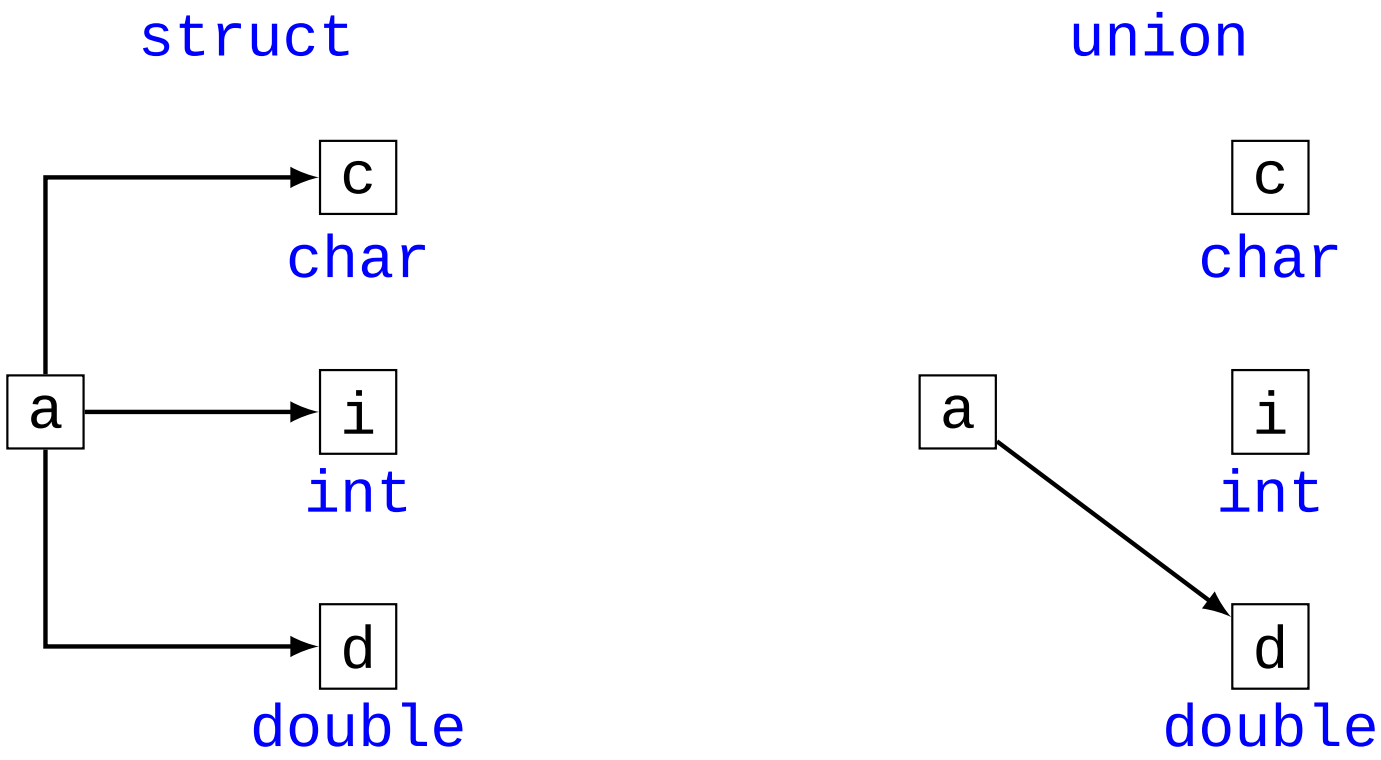
Unions

What are unions?



Unions

What are unions?



Unions

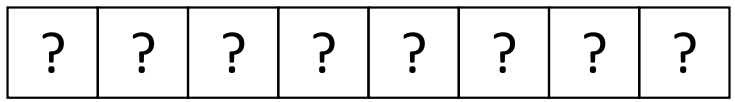
What are unions?

- *class types* consists of *classes*, *structs* and *unions*
- so a `union` is the final puzzle piece to fully define what class types are
- `struct` and `class` are practically identical (minor difference in terms of access specification). Specifically both of these construct allows us to bundle several values together, meaning each *data member* is stored and accessible simultaneously.
- however `union`s are quite different, because they only store *one* of the data members at a time.
- Specifically, the data members of unions are stored in the *same* memory, while each data member has its own memory in structs and classes.
- This means that the unions size is the same as the size of the largest data member.

Unions

Unions: memory model

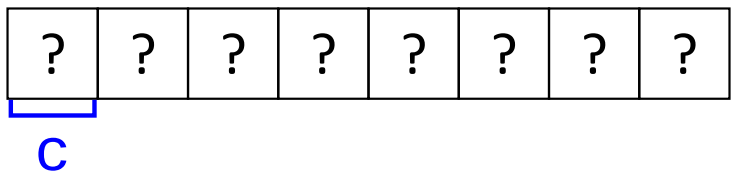
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

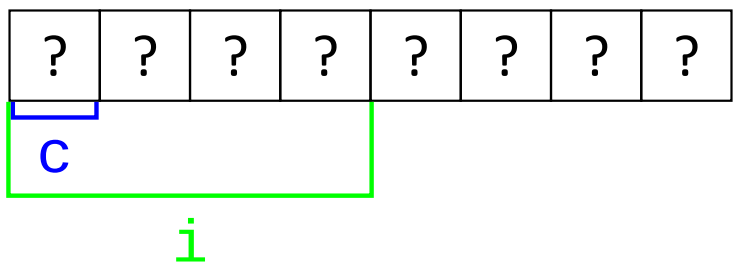
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

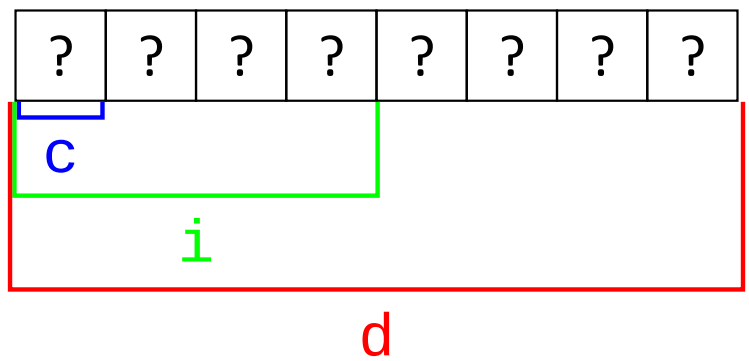
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Unions: memory model

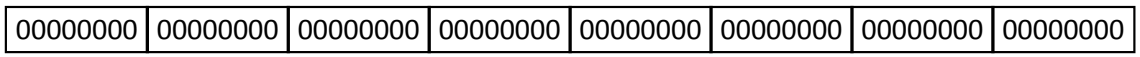
```
1  union My_Type
2  {
3    char c;   // 1 byte
4    int i;    // 4 bytes
5    double d; // 8 bytes
6  };
```



Unions

Using unions

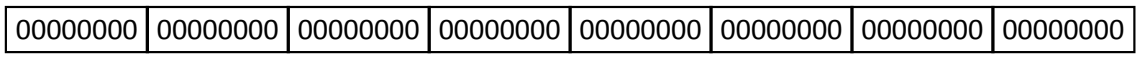
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```

a: 

Unions

Using unions

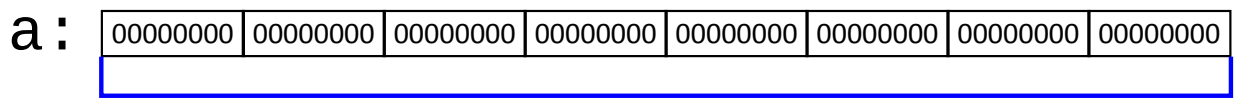
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```

a: 

Unions

Using unions

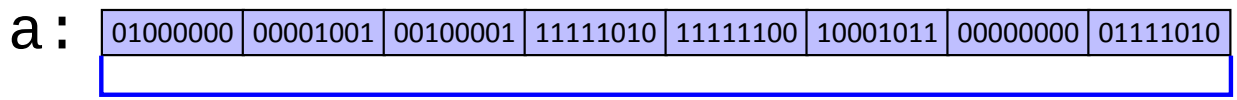
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

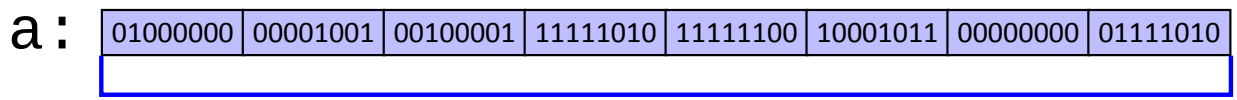
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

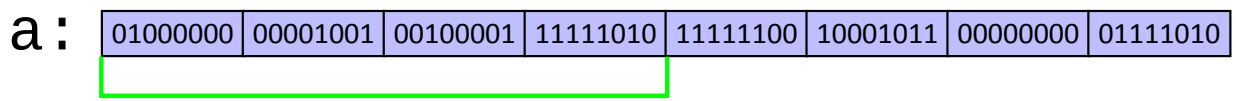
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

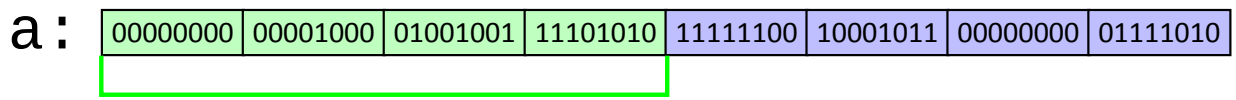
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

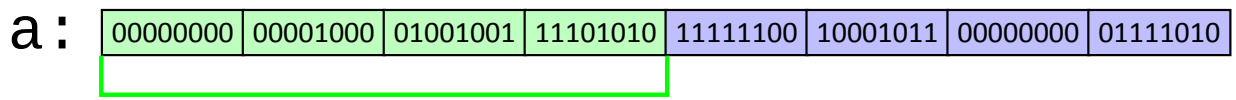
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

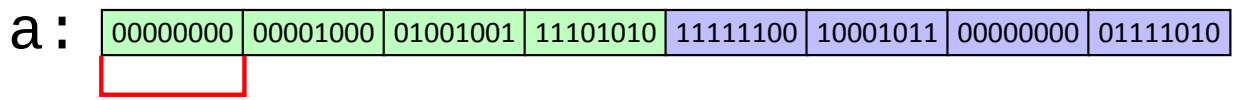
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

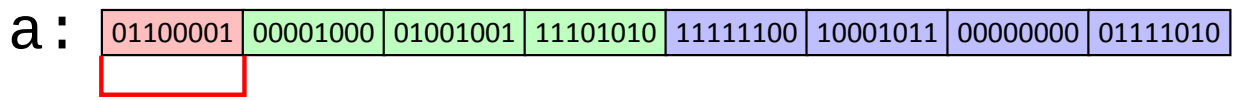
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



Unions

Using unions

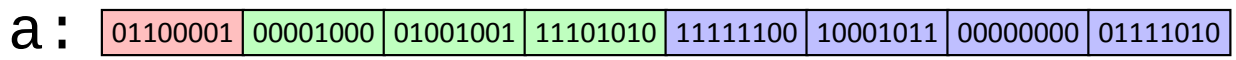
```
1  My_Type a;
2
3  a.d = 3.141592;
4  a.i = 543210
5  a.c = 'a';
6
7  std::cout << a.i << std::endl; // What happens?
```



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

Using unions

- Using unions is *syntactically* very similar to any other class type.
- However, one has to keep in mind that only *one* data member is valid at any point in time. We call this data member the *active* data member.
- Whenever we assign a value to a data member it puts it in its corresponding memory, ignoring that it will affect the other data members in some way.
- Neither the compiler nor the runtime keeps track of which data member is currently active so there is no way for us to check which is the active member.
- If we try to access a non-active data member it is strictly *undefined behaviour*.
- The byte/bit representation of data types can vary greatly between systems/platforms, so naturally the behaviour of accessing non-active members is not well defined across all of C++
- The *strict aliasing rule* can also cause a lot of problems with accessing non-active members (more on this later).

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

Error!

- The compiler says that it is unable to generate a default behaviour for the destructor of `My_Union`…
- Because the default implementation would be ill-formed...
- Since `std::string` has a non-trivial destructor...
- Why does that matter?
- Well, the string destructor should *only* be called if `text` is the active data member, otherwise we would attempt to call the destructor on a non-string member (a `double` value in this instance).
- Recall that the compiler nor the runtime actually knows which data member is active, so at the end-of-scope it is impossible for the compiler to specify what actually should happen.
- Let's ignore this, which will cause memory leaks, and see what happens!

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 }
```

u: 0.0

Unions

Non-trivial members

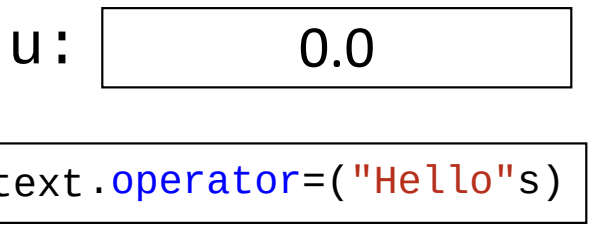
```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```

u: 0.0

Unions

Non-trivial members

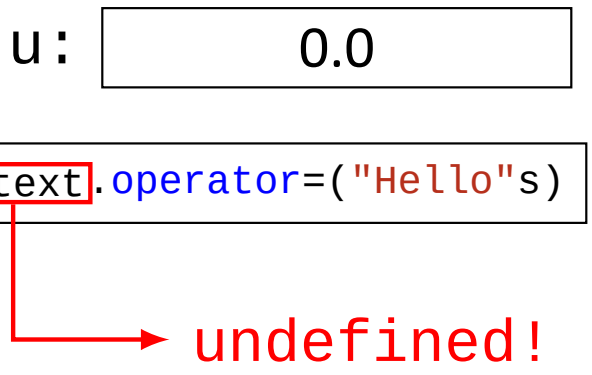
```
1  int main()
2  {
3    My_Union u { .value = 1.0 };
4    cout << u.value << endl;
5
6    u.text = "Hello"s;
7    cout << u.text << endl;
8
9    u.value = 3.14;
10   cout << u.value << endl;
11 }
```



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

Non-trivial members

- Note that the *assignment operator* assumes that the object we are assigning to *exists* and has been *initialized*.
- We can only initialize non-trivial objects in a constructor call.

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

Unsuccessful solution

- When changing the active member *from* a string, we are corrupting the string object by overwriting the bytes.
- This means that whatever assumptions `std::string::operator=()` makes are no longer met once the bytes have been changed.
- This is what causes the segmentation fault.
- This means that initializing the string when constructing the union is not enough to make the problem disappear.
- We would need to call the constructor of `std::string` again somehow before we actually assign to `text`…

Unions

How can we call the string constructor?

Unions

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

Unions

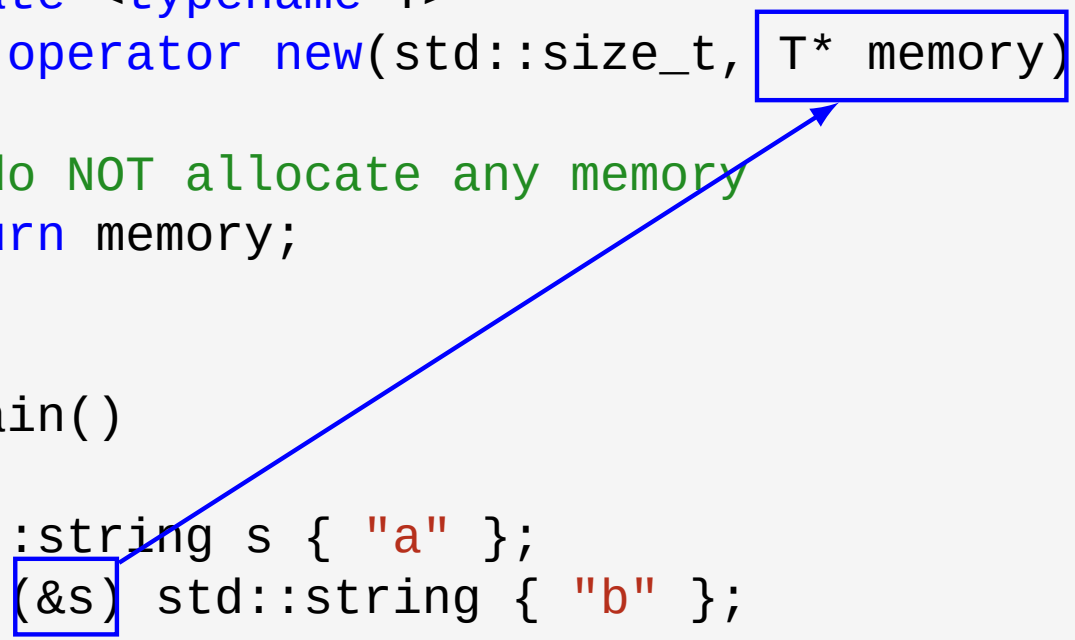
`operator new( )` with custom parameters

```
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

`operator new( )` with custom parameters

- We can define `operator new( )` with additional parameters. These are passed as additional arguments directly after the `new` keyword.
- You can therefore define an `operator new( )` with any number of custom parameters.
- In this case we are utilizing a key observation: `operator new( )` allocate memory and **calls the constructor**.
- So if we define an `operator new( )` which takes an address as a parameter and use that passed in memory as our “allocated” memory, then the effect is that the constructor will be called on the address.
- This operator is fortunately already defined in the STL (in the <new> header) and is called *placement new*.
- So named because it creates *new* objects *in-place* i.e. in already existing memory.

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

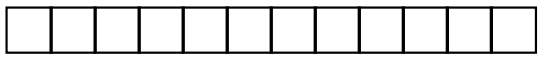
1. active member changes to non-trivial ⇒ use *placement new*
2. active member is already the type ⇒ use `operator=()`
3. active member changes from non-trivial ⇒ use `std::destroy_at()`

Can we make this easier?

Unions

Shared data members

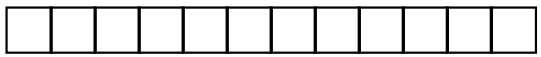
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

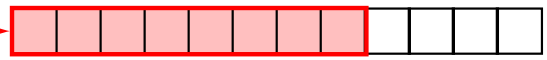
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

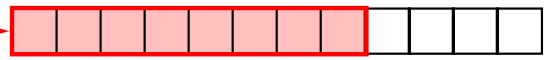
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

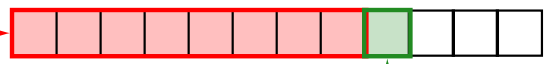
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

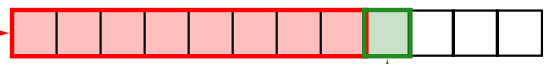
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

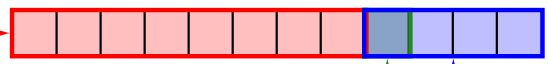
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

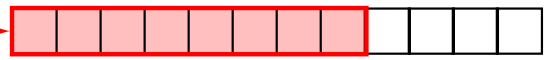
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



Unions

Shared data members

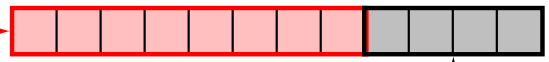
```
1  struct My_Struct
2  {
3    double d;
4    union
5    {
6      char c;
7      int i;
8    };
9  };
10
11 My_Struct m { .d = 0.5,
12                .c = 'a' };
```



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

Shared data members

- A `struct`/`class` can have data members that share memory (just like a `union`).
- This allows us to have some data members that are shared, and some that are separate and isolated.
- This is done by wrapping any data members that are supposed to be shared in a `union`-block *without* a name (if it is given a name, then it is interpreted as a `union` type definition inside the class).
- Using this, we can customize the behaviour of memory layout quite a lot.
- But it comes with all the complexities that `union` brings...

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

Strict aliasing

- Strict aliasing is a rule which allows for certain powerful optimizations.
- In particular, the compiler is allowed to assume that two objects of different types are *different* objects
- However, an exception to this rule is that you are *always* allowed to access individual bytes in *any* object through a `char*` and the compiler must respect, even though we are accessing parts of a T as if it was `char` objects.
- Note however, that the opposite is **not** true, i.e. we can access T objects as a `char*`, but we cannot access `char*` objects as a T*.

Unions

Anti-pattern: As-if rule and strict aliasing

- The reason it is dangerous to use the fact that union members share the same memory is because of *strict aliasing*
- When accessing individual members in the union, the compiler is allowed to *assume* that they are different objects, which leads to potential optimizations as we saw in the last code example.
- This usage of unions might be popular, but it is widely unsafe (it is *undefined behaviour*). Especially when combined with the optimizer of your compiler.
- This union usage is considered a (non-sanctioned) *hack* that works *sometimes*. There is no way to make this usage of unions guaranteed to be safe, since the behaviour varies from platform to platform, and strict aliasing + optimizers can cause problems.
- Either we have to make the compiler worse at optimizations when using unions (which means you potentially *lose* performance by introducing unions) *OR* we have to accept that this is not a valid use-case for unions. The C++ standard committee opted for the second option.

Unions

The variant pattern

- One drawback with `My_Union` is that the *user* have to make sure to call `My_Union::set_type()` each time they want to assign to a data member
- This makes the interface quite clunky to use and very error-prone (especially since no errors or warnings are generated whenever the user forgets to update the type properly)
- To fix this we design a *union-like* entity which we will call a `Variant` which ensures a cleaner and harder to misuse interface, but in essence solves the same problems as a union would.

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

Exercises

1. Make `My_Union` copyable and moveable
2. Use `std::type_index` and `typeid()` to keep track of types rather than `enum`
3. Implement all member functions of the `Variant` class introduced on the previous slide

Unions

Generalization of `Variant`

```
template <typename... Ts>
class Variant
{
public:
  // ...
  template <typename T>
  T& get();

  template <typename T>
  bool has() const;

  template <typename T>
  Variant& operator=(T const& value);

private:
  // how to store???
};
```

Unions

Generalization of `Variant`

- Of course, the `Variant` class we have introduced can only store strings and doubles, but in a more general context we would like to allow the user to decide which – and how many – types should be possible to store in the object.
- One initial step to this is to redesign the interface with variadic templates and general template `get()`, `has()` and `operator=()` functions.
- But now the question becomes: how do I generate a `union` based on `Ts`?
- The unfortunate news is that in current versions of C++ we cannot do that.
- We are going to have to dig even deeper into the complexities of memory to se how we can emulate the behaviour of unions without actually using `union`.

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?

But then we must save it...

ptr

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

How to fix this?

- The optimizer tracks objects throughout the compilation so that it can identify wheter we read/write from/to the object so that it can deduce what “level” of optimization can be done on the object.
- Here the tracked object is `memory`, which is just a `char`-array, so even though we *somewhere* transfered an object of type `T` into `memory`, the optimizer sees this as us modifying the array, but it does not *understand* (due to strict aliasing) that the bytes in `memory` corresponds to a `T` object.
- But *we* know that there actually is a `T` object stored at `memory`, we’ve just created it with *placement new* and got a completely valid pointer to it (`ptr`). However, we have *lost* that pointer so from the compilers point-of-view there is no valid way to find it anymore.
- We need some type of mechanism to tell the compiler to *not* try to track where an object came from (i.e. disable certain optimizations) and just trust us (on blind faith) that the accessed object is *valid*.
- In C++17 this was introduced, it is called `std::launder()`!

General-purpose storage

Is this a valid way of accessing the data?

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

General-purpose storage

- `std::launder()` has no direct runtime cost (although it does disable some optimizations, so it might implicitly have some effect on the generated code)
- It is still undefined behaviour to `std::launder()` pointers into objects that has not actually been created there. I.e. we are saved here by the fact that at *some point* placement new was used to actually construct an object. The first implementation we showed of `transfer()` would *not* save us in this situation.
- `std::launder()` just tells the compiler to not worry about where the object/memory actually came from, this does not mean we are free of any consequences, because we are essentially telling the compiler to trust us, and usually the compiler knows better than we do so if the compiler can't trust us we will end up with inconsistent and undefined programs.
- A good resource to understand `std::launder()` is this stackoverflow thread: https://stackoverflow.com/questions/39382501/what-is-the-purpose-of-stdlaunder

General-purpose storage

Putting it together (somewhat)

```
template <typename... Ts>
class Variant {
public:
  // ...
  template <typename T>
  T& get() {
    return *std::launder(
      reinterpret_cast<T*>(&memory[0])
    );
  }
private:
  char memory[max_size<Ts...>];
};
```

General-purpose storage

Exercise

This week's exercise is to complete the implementation of `Variant`

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

Non-global allocation

Introduction

- So far we have seen how to replace the global allocation scheme, but this is often times overkill
- Many times the need is often more geared towards handling certain types of objects
- This need will be demonstrated on the next slide

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

Inefficient allocation pattern

- In, for example, particle simulations or games etc. it is quite common that we have objects that are constantly destroyed and created.
- Many such applications need high performance, even during particularly intense simulation steps (i.e. steps where many objects are destroyed and/or created).
- To avoid having to deallocate a number of objects followed by a series of allocations, what if we reuse the objects?
- This can of course be done “manually” but then the performance breaks the second someone doesn’t use the intended functionality.
- Therefore it is probably better to simply overload the behaviour of `new` and `delete`, but *only* for the `Particle` objects, so that it reuses previous allocations if possible.

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

Reusing allocations if possible

- The easiest (but arguably not the most efficient) way to reuse memory is to keep a so-called *free list*, i.e. a list of all available allocations that are currently unused.
- We want to keep such a list for *all* particle allocations in the system, so we make the list a static variable in `Particle` (however we make it private so it is unavailable outside of the allocations).
- After that we can declare (and implement) `operator new()` and `operator delete()` as static member functions of `Particle`. The effect of this is that whenever the user writes `new Particle` or deletes a particle through a pointer, it will call the specialized version of the operators, rather than the globally available one.
- This can however lead to interferences in the global allocation scheme (for example here we are forcing this part of the program to use `std::malloc()` which might not be the allocation scheme used globally.
- To fix this we instead call the global `operator new()`.

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

A different scenario: Discussion

- We assume that `process()` is called *often* and should be reasonably fast as to avoid interfering in other tasks.
- We can for example see this type of pattern when collecting statistics and data about a simulation.
- The pattern we are studying here is the fact that we have some *persistent* data (the `results` vector) which is calculated from temporary data (the `values` vector).
- The problem here is that the persistent data should be allocated *as normal*, but if we decide to do the same for the temporary data we will potentially do *many* allocations that are deallocated shortly after.
- Here we would benefit from the `values` allocations being reused, but we *DO NOT* want to do the same for all vectors.

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

Allocators

- Allocators is really a design pattern used by the STL to allow for customized memory management of individual containers.
- It allows us to pick-and-choose how memory is managed for each container without affecting the global allocation strategies.
- Note that this is a rich and complex topic, so this lecture will only introduce the idea, but beware that allocators and memory management is a huge topic and fairly important within certain industries.
- Allocators are commonly used when performance matters a lot, or when memory is significantly constrained.

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