Sweating the Small Stuff: Brace Initialization, Unions, and Enums

By Scott Schurr for C++ Now 2013

Topics

- Brace Initialization
- Unions
- Enums

Brace Initialization

Executive Summary:

- When initializing, usually prefer using brace initializers without an equals sign.

- In class design, be wary of collisions between constructors that take `std::initializer_list<>` and those that don't.

Initialization

- Gives an object its **initial** value.
- Initialization is not assignment.
  - E.g., const objects can be initialized but not assigned.

```
const int x = 4;          // Initialization
x = 5;                    // Assignment to const [error]
int y = 6;                // Initialization
y = x;                    // Assignment to non-const [fine]
```

Initialization in C++98...

Many ways to initialize...

```
const int a = 5;                    // "copy initialization"
const int b(6);                     // "direct initialization"
const int c = {7};                  // brace initialization
const int c_arr[] = {7, 8, 9};      // brace initialization
struct S1 { int x; int y; };
S1 s1a = {10, 11};                  // brace initialization
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a(12, 13);                     // function call syntax
```

...Copy Initialization...

Many ways to initialize...

```
const int a = 5;                   // "copy initialization"
const int b(6);                    // "direct initialization"
const int c = {7};                 // brace initialization
const int c_arr[] = {7, 8, 9};     // brace initialization
struct S1 { int x; int y; };
S1 s1a = {10, 11};                 // brace initialization
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a(12, 13);                    // function call syntax
```

...Direct Initialization...

Many ways to initialize...

```
const int a = 5;                  // "copy initialization"
const int b(6);                   // "direct initialization"
const int c = {7};                // brace initialization
const int c_arr[] = {7, 8, 9};    // brace initialization
struct S1 { int x; int y; };
S1 s1a = {10, 11};                // brace initialization
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a(12, 13);                   // function call syntax
```

…Brace Initialization…

Many ways to initialize…

```
const int a = 5;                    // "copy initialization"
const int b(6);                     // "direct initialization"
const int c = {7};                  // brace initialization
const int c_arr[] = {7, 8, 9};      // brace initialization
struct S1 { int x; int y; };
S1 s1a = {10, 11};                  // brace initialization
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a(12, 13);                     // function call syntax
```

…Function Call Syntax

Many ways to initialize…

```
const int a = 5;                  // "copy initialization"
const int b(6);                   // "direct initialization"
const int c = {7};                // brace initialization
const int c_arr[] = {7, 8, 9};   // brace initialization
struct S1 { int x; int y; };
S1 s1a = {10, 11};                // brace initialization
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a(12, 13);                   // function call syntax
```

More Initialization in C++98

Hmmm… Do I use ( ) or { }?

```
const int c_arr[] = (7, 8, 9); // error
struct S1 { int x; int y; };
S1 s1a = (10, 11);             // error
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a = {12, 13};             // error
```

Sometimes It Is Awkward

- Containers require another container.

```
const int values[] = { 2, 3, 5, 7, 11 };
const std::vector<int> v(values, values+5);
```

Sometimes It Is Awkward

- Containers require another container.

```
const int values[] = { 2, 3, 5, 7, 11 };
const std::vector<int> v(values, values+5);
```

- Member and new arrays are impossible.

```
class BadNews {
public:
    explicit BadNews() : data(???) { }
private:
    int data[5];                        // not initializable
};
const int* pData = new const int[4]; // not initializable
```

Aside: Narrowing Conversions

A conversion is narrowing if

- The target value can't exactly represent all possible source type values, and
- The compiler can't guarantee that the target type can hold the source value.

```
int x = 2.0f;      // float to int is always narrowing
float f = x;       // narrowing for any int > 16777215
unsigned u1 = x;   // signed to unsigned is narrowing
unsigned u2 = 25; // not narrowing.  25 is representable.
```

Implicit Narrowing in C++98

- C++98 allows implicit narrowing in brace initialization [C++ Standard, 2003 Section 8.5.1 paragraph 12].

```
const int arr[] = { 0.9, 1.9, 2.9 }; // Okay in C++98

for (std::size_t i = 0;
    i < sizeof(arr) / sizeof(arr[0]); ++i)
{
    assert(i == arr[i]);                // Asserts don't fire
}
```

Implicit Narrowing in C++11

- C++11 forbids implicit narrowing in brace initialization [FDIS section 8.5.4 paragraph 2].

```
const int arr[] = { 0.9, 1.9, 2.9 }; // Compile error
```

- This is a breaking change…
- But it is code that deserves to be broken.

C++11 Initialization

- C++11 **adds** yet another type of initialization: brace with no equals

```
int x { 5 };
```

- All other initialization forms still work.

Use New Form Everywhere

```
const int a { 5 };
const int c_arr[] {7, 8, 9};

struct S1 { int x; int y; };
S1 s1a {10, 11};

struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 s2a {12, 13};                    // Calls S2 constructor
```

…Even Where C++98 Couldn't

- Container initialization.

```
const std::vector<int> v { 2, 3, 5, 7, 11 };
```

- Member and `new` array initialization.

```
class GoodNews {
public:
    GoodNews() : data { 5, 4, 3, 2, 1 } { }
private:
    int data[5];
};
const int* pData { new const int[4] { 1, 2, 3, 4 } };
```

…And Some Surprising Places

- Return values.

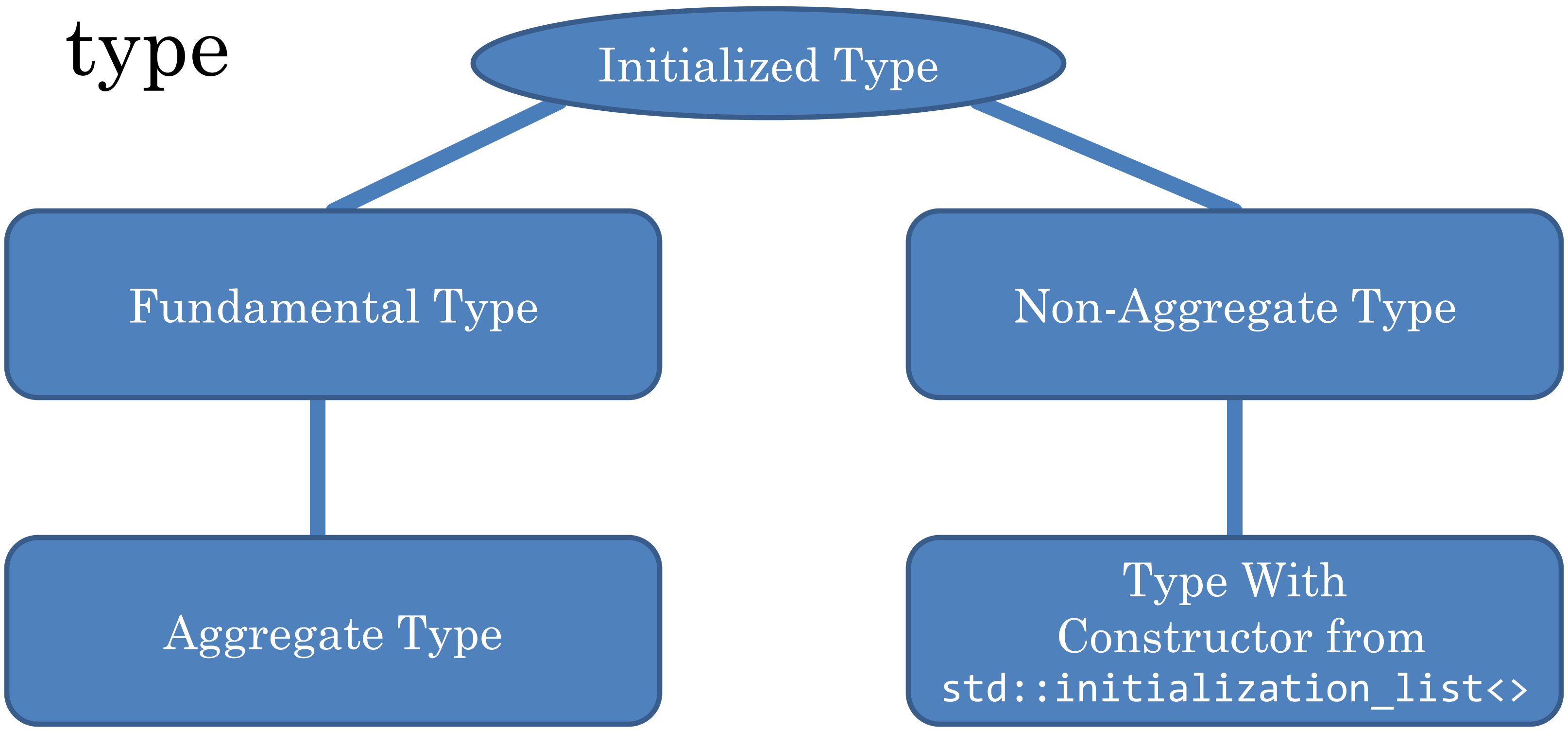
```
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
S2 MakeS2() { return { 0, 0 }; }  // calls S2 constructor
```

- Function call parameters.

```
void useVector(std::vector<int>); // function declaration
useVector( { 0, 1, 2, 3, 4 } );   // function invocation
```

Brace Initialization Tour

- The rules are based on the initialized type



Aside: Aggregate Types

- From C++11 FDIS 8.5.1 paragraph 1:

  An *aggregate* is an array or a class with no user-provided constructors, no initializers for non-static data members, no private or protected non-static data members, no base classes, and no virtual functions.


20 mm Aggregates

Aggregate Examples

- Aggregates

```
int arr[5];
```

Aggregate Examples

- Aggregates

```
int arr[5];
struct S1 { int x; int y; };
```

Aggregate Examples

- Aggregates

```
int arr[5];
struct S1 { int x; int y; };
class C1 {
public:
    int x; int y;
};
```

Aggregate Examples

- Aggregates

```
int arr[5];
struct S1 { int x; int y; };
class C1 {
public:
    int x; int y;
};
std::array<int, 5> std_arr;   // FDIS 23.3.2.2 para 1
```

Aggregate Examples

- Aggregates

```
int arr[5];
struct S1 { int x; int y; };
class C1 {
public:
    int x; int y;
};
std::array<int, 5> std_arr;   // FDIS 23.3.2.2 para 1
```

- Not Aggregates

```
struct S2 {                   // Why?
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
```

Aggregate Examples

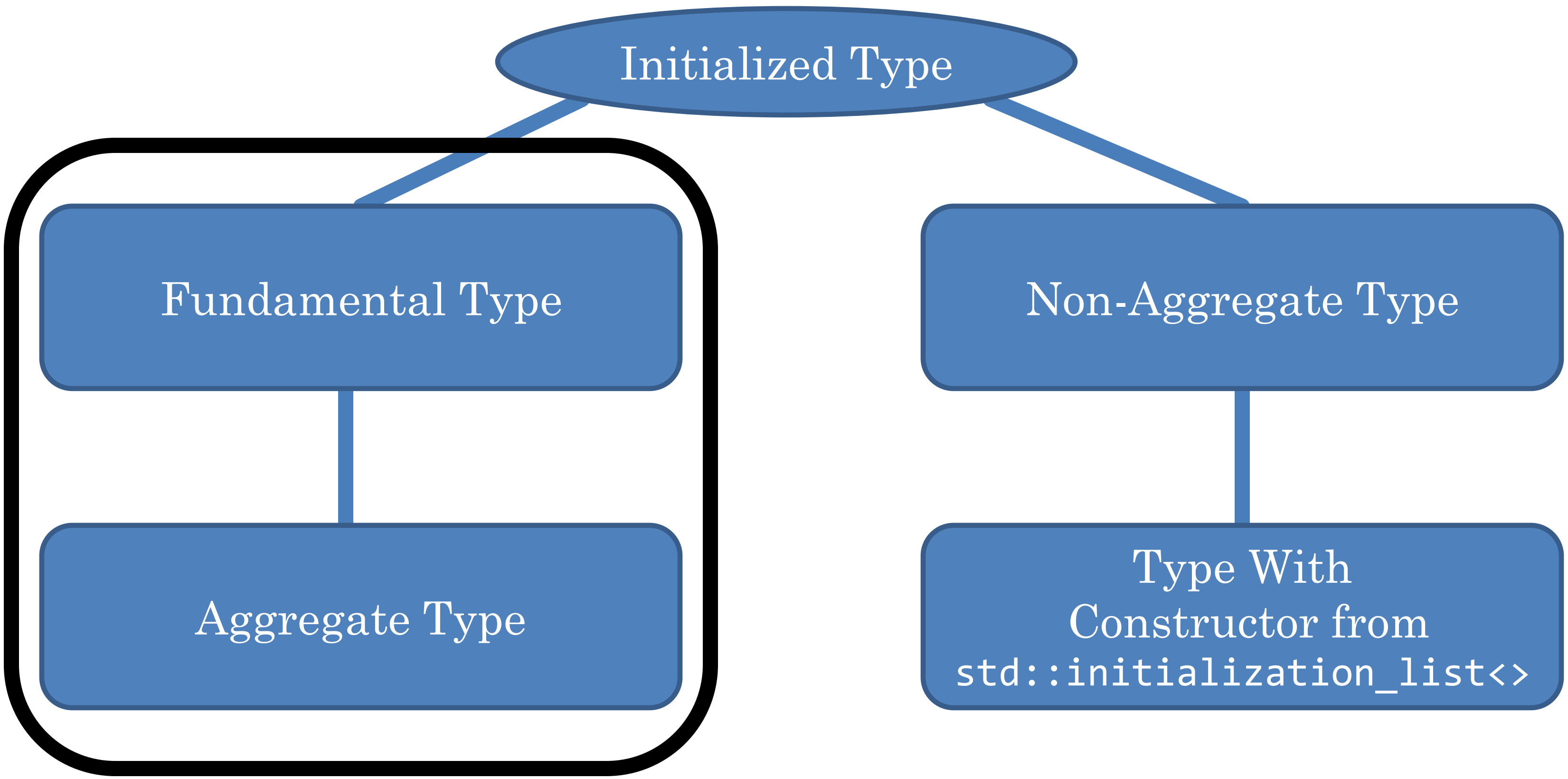
- Aggregates

```
int arr[5];
struct S1 { int x; int y; };
class C1 {
public:
    int x; int y;
};
std::array<int, 5> std_arr;    // FDIS 23.3.2.2 para 1
```

- Not Aggregates

```
struct S2 {
    int x; int y;
    S2(int arg1, int arg2) : x(arg1), y(arg2) { }
};
class C2 : public C1 { };    // Why?
```

Brace Initializing Fundamental and Aggregate Types



Brace Initializing Fundamental and Aggregate Types

- Elements are assigned beginning-to-end.

```
int x { 3 };
int arr[] { 4, 5, 6, 7 };    // arr[0] == 4

struct S1 { int x; int y; };
S1 s1a { 8, 9 };             // s1a.x == 8; s1a.y == 9
```

- Note that `std::array<>` holds a single element. That element holds an array.

```
std::array<int, 3> stdArray { { 1, 2, 3 } };
```

Brace Initializing a Union

- The first element of a union is assigned.

```
union U1 {
    const char* p;              // <- assigns p
    int i;
};

U1 u1 { "Hi Mom!" };            // okay
U1 u2 { 0 };                    // 0 is a valid pointer
U1 u3 { 1 };                    // error doesn’t compile
```

Initializing Aggregate Types

- Members are assigned beginning-to-end.
- Too many arguments: error

```
float f_arr[3] { 0.1f, 0.2f, 0.3f, 0.4f };  // error
```

- Too few arguments…
  - Fundamental types: zeroed [or nullptr]
  - UDT without constructor: members zeroed
  - UDT with default constructor: defaulted
  - UDT with **no** default constructor: error

Examples of Too Few Arguments

- UDT with no constructor: members zeroed.

```
struct Inner1 { int a; int b; };
struct Outer1 { Inner1 c; Inner1 d; };
Outer1 o1 { { 42, 99 } };   // o1.d.a == 0; o1.d.b == 0
```

Examples of Too Few Arguments

- UDT with no constructor: members zeroed.

```
struct Inner1 { int a; int b; };
struct Outer1 { Inner1 c; Inner1 d; };
Outer1 o1 { { 42, 99 } };   // o1.d.a == 0; o1.d.b == 0
```

- UDT with default constructor: defaulted.

```
struct Inner2 { int x_; Inner2() : x_ { 7 } { } };
struct Outer2 { Inner2 y; };
Outer2 o2 { };                // o2.y.x_ == 7
```

Examples of Too Few Arguments

- UDT with no constructor: members zeroed.

```
struct Inner1 { int a; int b; };
struct Outer1 { Inner1 c; Inner1 d; };
Outer1 o1 { { 42, 99 } };   // o1.d.a == 0; o1.d.b == 0
```

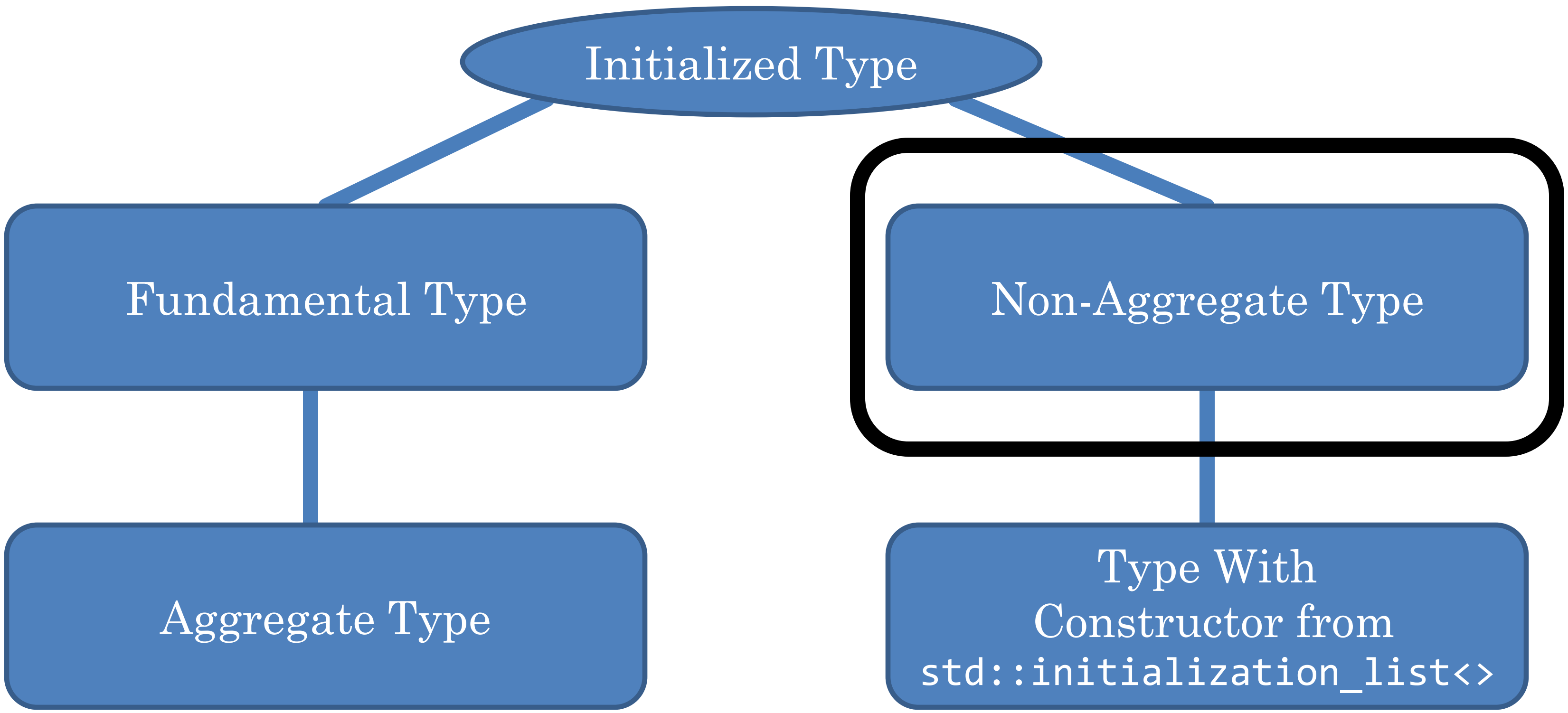
- UDT with default constructor: defaulted.

```
struct Inner2 { int x_; Inner2() : x_ { 7 } { } };
struct Outer2 { Inner2 y; };
Outer2 o2 { };              // o2.y.x_ == 7
```

- UDT with **no** default constructor: error.

```
struct Inner3 { int r_; Inner3(int arg) : r_ { arg } { } };
struct Outer3 { Inner3 s; };
Outer3 o3 { };              // error won't compile
```

Brace Initializing Non-Aggregates



Initializing Non-Aggregates

- Invokes a constructor.

```
struct S3 {
    const int x;
    explicit S3()             : x { 0 } { } // default ctor
    explicit S3(int a)        : x { 1 } { } // 1 arg ctor
             S3(int a, int b) : x { 2 } { } // 2 arg ctor
};
S3 s3a;                  // s3a.x == 0
S3 s3b { };              // s3b.x == 0
S3 s3c { 0 };            // s3c.x == 1
S3 s3d { 0, 0 };         // s3d.x == 2
```

Empty Braces Invoke the Default Constructor

```
class NoDefault {
    const int x_;
public:
    NoDefault(int x) : x_ {x} { };
};

NoDefault nd1 { };          // error – no default ctor
```

Non-Aggregates and Narrowing

- C++11 constructor calls using parentheses perform implicit narrowing [C++98 compatible].

```
struct S4 {
    int x;
    explicit S4(int a) : x {a} { }
};
S4 s4a(2.0f);                      // narrowing is bad
```

Non-Aggregates and Narrowing

- C++11 constructor calls using parentheses perform implicit narrowing [C++98 compatible].

```
struct S4 {
    int x;
    explicit S4(int a) : x {a} { }
};
S4 s4a(2.0f);                   // narrowing is bad
```

- Construction with braces does not narrow.

```
S4 s4b { 2.0f };                // compile-time error (good!)
```

Non-Aggregates and Narrowing

- C++11 constructor calls using parentheses perform implicit narrowing [C++98 compatible].

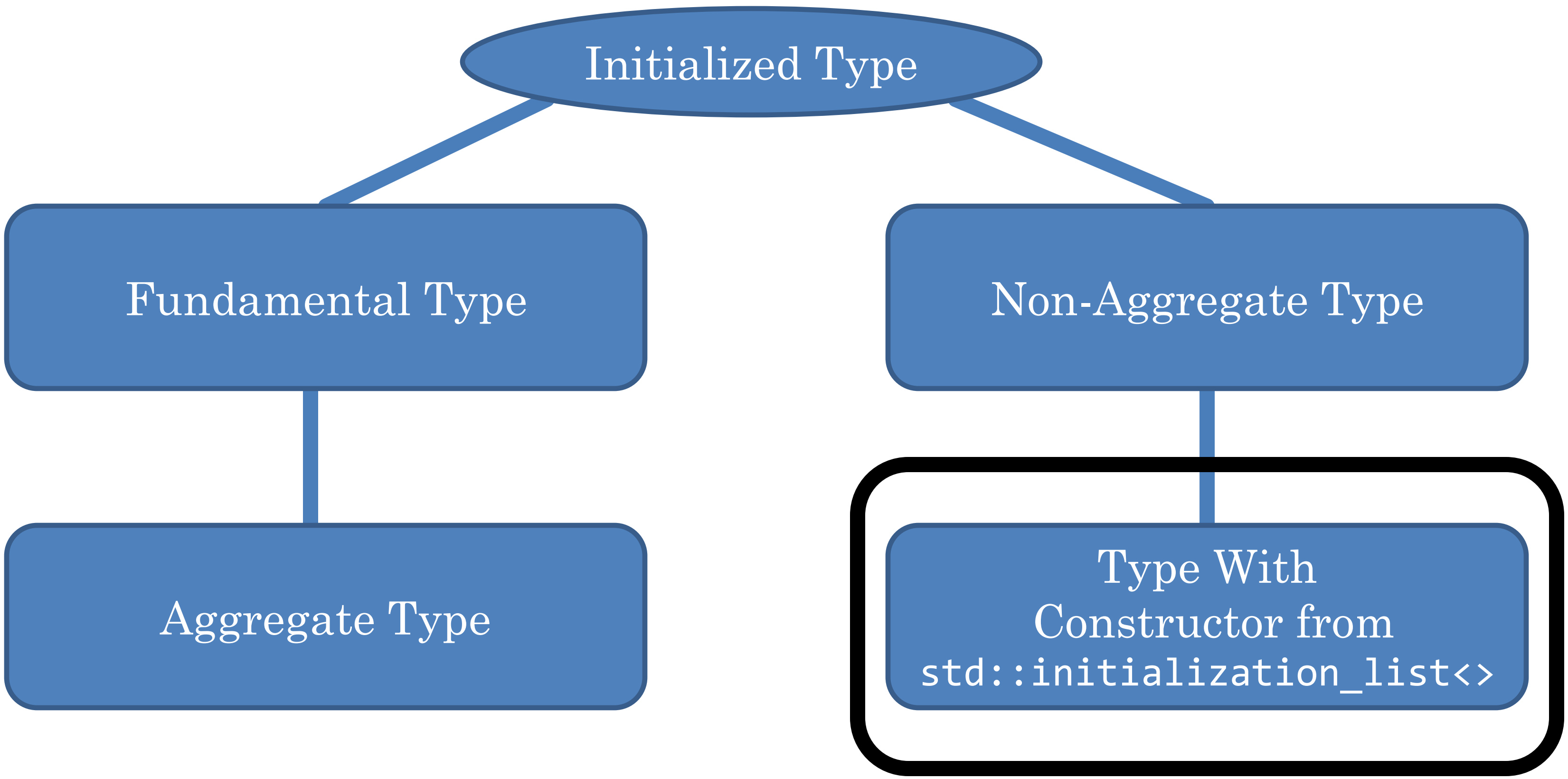
```
struct S4 {
    int x;
    explicit S4(int a) : x {a} { }
};
S4 s4a(2.0f);                   // narrowing is bad
```

- Construction with braces does not narrow.

```
S4 s4b { 2.0f };                // compile-time error (good!)
```

- Usually prefer brace initialization.

Brace Initializing Non-Aggregate Containers



Brace Initializing Non-Aggregate Containers

To initialize the contents of containers use

```
std::initializer_list<>
```

```
vector(initializer_list<T>, const Allocator& = Allocator());

list(initializer_list<T>, const Allocator& = Allocator());

map(initializer_list<value_type>,
     const Compare& = Compare(),
     const Allocator& = Allocator());
```

Brace Initializing Containers

```
const int i { 3 };
int f() { return 6; }

// Simple container
std::vector<int> v { 7, i, 2 * i, f(), f() + i };

// More interesting container
std::map<int, const char*> m { {i, "a"}, {f(), "b"} };
```

- A `std::initializer_list<T>` is uniform. Otherwise there are no uniformity requirements for brace initialization.

`std::initializer_list<>`

`#include <initializer_list>` is magic!

- It allows the compiler to convert

```
{ T a, T b }
```

- into

```
std::initializer_list<T>
```

More `std::initializer_list<>`

- `initializer_list<>` stores its values in an array.

More `std::initializer_list<>`

- `initializer_list<>` stores its values in an array.

- The underlying array is not guaranteed to exist beyond the `initializer_list<>`.

More `std::initializer_list<>`

- `initializer_list<>` stores its values in an array.

- The underlying array is not guaranteed to exist beyond the `initializer_list<>`.

- `initializer_list<>` has 3 methods:
  - `size()        // number of elements in array`
  - `begin()       // pointer to start of array`
  - `end()         // pointer to one past end of array`

Constructor Example

```
class C3 {
    int store_[5];
public:
    explicit C3(int value)
        { for (int& n : store_) n = value; }
    explicit C3(std::initializer_list<int> values)
    {
        const int* v_ptr = values.begin();
        for (std::size_t i = 0; i < 5; ++i) {
            store_[i] = i < values.size() ? *v_ptr++ : 0;
        }
    }
    int operator[](size_t i) const { return store_[i]; }
};
```

Overloading

```
class C3 {
    int store_[5];
public:
    explicit C3(int value);
    explicit C3(std::initializer_list<int> values);
    ...
};
```

Brace lists prefer initializer_list overloads.

```
C3 c3a { 5 };   // explicit C3(std::initializer_list<int>)
C3 c3b ( 5 );   // explicit C3(int)
```

You can't *always* use brace initialization!

Further Overload Examples

- Examples of this concern can also be found in the standard library…

```
std::list<int> list1 { 7 }; // list1.size() == 1
std::list<int> list2 ( 7 ); // list2.size() == 7
```

- Not an issue for list<T> where T != int.

- Know which overload you get!

Yet More Overload Examples

- Conversion rules determine the choice between `initializer_list<>` overloads.

```
class C4 {
public:
    explicit C4 (std::initializer_list<int> rhs);
    explicit C4 (std::initializer_list<double> rhs);
};
C4 c4a { 0.0f, 1.5f, 3.0f };  // float to double wins
C4 c4b { 1, 2.0, 3 };         // error ambiguous
long double ld { 7.5 };
C4 c4c { ld, ld, ld };        // error narrowing
```

`std::initializer_list<>` Lifetime

- The data in an `initializer_list<>` may not outlive the `initializer_list<>`.

```
void fill_vector(std::vector<const int*>& vec)
{
    std::initializer_list<int> fill {2, 3, 5};
    for (const int& i : fill) { vec.push_back(&i); } // !
    return;
}
```

- Don't keep pointers or references to `initializer_list<>` contents.

`initializer_list<>` Deduction

- An `initializer_list<>` can be templatized.

```
class C5 {
public:
    template <typename T>
    explicit C5 (std::initializer_list<T> rhs);
    ...
};
C5 c5a {2.0f, 3.5f, 5.0f}; // T == float
```

- A non-uniform `initializer_list<>` cannot be deduced.

```
C5 c5b {2.0f, 3.5, 5};     // error... what is T?
```

initializer_list<> and auto

- Consider...

```
auto i { 1 };
```

- What is the type of i? Is it int?

```
static_assert(                          // Assert does not fire
    std::is_same< decltype(i),
    std::initializer_list<int> >::value, "Yipes!");
```

- Surprise! auto always brace initializes as std::initializer_list<>.

Other `initializer_list<>` Uses

A `std::initializer_list` can be a parameter of any function, not just constructors…

```
vector<T>& vector<T>::operator=(initializer_list<T>)
void vector<T>::assign(initializer_list<T>);
iterator vector<T>::insert(const_iterator position,
   initializer_list<T> il);
```

But the subject is initialization, so let's not get distracted.

Brace Initialization Summary

- When initializing, usually prefer using brace initializers without an equals sign.

- In class design, be wary of collisions between constructors that take `std::initializer_list<>` and those that don't.

Unions

Executive Summary:

- Always know what you put in your unions.

- Unions can now hold non-POD types. They require placement new and explicit destruction.

- Become aware of anonymous unions.

- Consider unions for placement new.


Union

What’s a Union? Pt 1

- It looks like a struct, but…
- All its elements are stored at the same location.

```
struct S1 { char x; double y; };
static_assert(offsetof(struct S1, x) !=
              offsetof(struct S1, y), "Weird struct");

union U1 { char x; double y; };
static_assert(offsetof(union U1, x) ==
              offsetof(union U1, y), "Weird union");
```

What's a Union? Pt 2

- A union is only big enough to hold its largest element.

```
struct S1 { char x; double y; };
static_assert(sizeof(S1) >=
    sizeof(S1::x) + sizeof(S1::y), "Weird struct");

union U1 { char x; double y; };
static_assert(sizeof(U1) == sizeof(U1::y), "Weird union");
```

- So a union can hold just one of its elements at a time.

Access Control in Unions

- Union members can have access control.

```
union UPrivate {
private:
    char c;
public:
    int i;
};
UPrivate up1;
up1.c = '%';                    // error
up1.i = 42;
```

- By default union members are public.

A Union Can Hold Structs

```
enum UType { e_char, e_double };
struct S2C { UType t; char c; };
struct S2D { UType t; double d; };

union U2 { S2C s2c; S2D s2d; };
```

- The union contains one struct at a time.

```
S2C s2c { e_char, '!' };
S2D s2d { e_double, 3.1415926 };
U2 u2;
u2.s2c = s2c;
u2.s2d = s2d;
assert(u2.s2c.t == e_double);
```

Hah! You Cheated!

```
struct S2C { UType t; char c; };
struct S2D { UType t; double d; };
S2C s2c { e_char, '!' };
S2D s2d { e_double, 3.1415926 };
U2 u2;
u2.s2c = s2c;
u2.s2d = s2d;
assert(u2.s2c.t == e_double);
```

The assigned member is not the same as the examined member

Not Cheating

```
struct S2C { UType t; char c; };
struct S2D { UType t; double d; };
S2C s2c { e_char, '!' };
S2D s2d { e_double, 3.1415926 };
U2 u2;
u2.s2c = s2c;
u2.s2d = s2d;
assert(u2.s2c.t == e_double);
```

Common initial sequence

The assigned member is not the same as the examined member

- The standard allows this comparison because both structs have a “common initial sequence” [FDIS 9.2 para 20].

Union Initialization

- Given...

```
enum UType { e_char, e_double };
struct S2C { UType t; char c; };
struct S2D { UType t; double d; };
union U2 { S2C s2c; S2D s2; };
```

- Indeterminate initial value

```
U2 u2a;                                  // u2a.s2c.t == ???
```

- External brace initialize **first** union member

```
U2 u2b {{e_char, '!' }};                 // okay
U2 u2c {{e_double, 2.71828182 }}; // error
```

More Union Initialization

- Internal brace initialize any **one** member

```
union U3 {
    S2C s2c;
    S2D s2d {e_double, 1.732050};
};
U3 u3;                          // u3.s2c.t == e_double
```

- Provide constructor(s)

```
union U4 {S2C s2c; S2D s2d;
    U4(char rhs)   { s2c.t = e_char;   s2c.c = rhs; }
    U4(double rhs) { s2d.t = e_double; s2d.d = rhs; }
};
U4 u4{1.414213562};             // u4.s2d.t == e_double
```

Union Member Functions

- A union can have member functions…
- …including constructors and destructors.
- But no virtual functions! Why?
  - Because a union can have only one active member at a time. The union has no place to keep a virtual table pointer.
- A union may not have a base class.
- A union cannot be a base class.

C++98: A Union Cannot Hold…

- An object with non-trivial constructor,
- Or with non-trivial copy assignment,
- Or with non-trivial destructor,
- Or an object with any virtual member.
- An array of any of the above.
- A static data member.
- A reference type.
- A virtual member function.

C++11: A Union Cannot Hold…

- ~~An object with non-trivial constructor,~~
- ~~Or with non-trivial copy assignment,~~
- ~~Or with non-trivial destructor,~~
- ~~Or an object with any virtual member.~~
- ~~An array of any of the above.~~
- ~~A static data member.~~
- A reference type.
- A virtual member function.

Non-PODs in Unions

"In general, one must use explicit destructor calls and placement new operators to change the active member of a union." [FDIS Section 9.5 para 4]

Some Perspective

- A union is less dangerous than a `reinterpret_cast<>` or c-style cast.
  - Contained types are explicit.
  - Controlled access.

- Occasionally we need tools like this.

- It's why we love C++!

The Compiler to the Rescue

In C++11 every union starts with the following implicit member functions:

Union

- Implicit Default Ctor
- Implicit Copy Ctor
- Implicit Move Ctor
- Implicit Copy Assign
- Implicit Move Assign
- Implicit Destructor

Implicitly Deleted Functions

If **any** member of the union has any non-trivial implementation of any of these member functions, those functions are implicitly deleted from the union.

Union	Union Member
~~Default Ctor = delete~~	Non-triv Default Ctor
Implicit Copy Ctor	Trivial Copy Ctor
Implicit Move Ctor	Trivial Move Ctor
Implicit Copy Assign	Trivial Copy Assign
Implicit Move Assign	Trivial Move Assign
~~Destructor = delete~~	Non-triv Destructor

Replacing Deleted Functions

But any of these implicitly deleted member functions may be user-provided

Union

- User-def Default Ctor
- Implicit Copy Ctor
- Implicit Move Ctor
- Implicit Copy Assign
- Implicit Move Assign
- User-def Destructor

Union Member

- Non-triv Default Ctor
- Trivial Copy Ctor
- Trivial Move Ctor
- Trivial Copy Assign
- Trivial Move Assign
- Non-triv Destructor

Let's Try It

```
struct Place {
    int x_; int y_;
    explicit Place() : x_{0}, y_{0} { }
    explicit Place(int x, int y) : x_{x}, y_{y} { }
};
union Stuff1 {
    int i;
    double d;
    Place p;
};

Stuff1 s1a;       // error - deleted default ctor
Stuff1 s1b{0};    // Okay. Aggregate initializes i
```

Let’s Fix Stuff

```
struct Place {
    int x_; int y_;
    explicit Place() : x_{0}, y_{0} { }
    explicit Place(int x, int y) : x_{x}, y_{y} { }
};
union Stuff2 {
    int i;
    double d;
    Place p;
    Stuff2() : p{0, 0} { }  // User-defined default ctor
};
Stuff2 s2a;       // Calls user supplied default ctor
Stuff2 s2b{0};    // error - Stuff2 is not an aggregate
```

Anonymous Unions

```
class C6 {
private:
    union {                      // <- No name
        int i_;
        double d_;
    };
public:
    C6 () : d_ { 0.0 } { }
    ...
};
```

i_ and d_ are in an anonymous union.

Used like normal members (no extra name).

Anonymous Union Rules

- No private or protected members.
- No static members.
- No member functions.
- If at global or namespace scope, must be static.
- A union for which objects or pointers are declared is not anonymous.

```
union { int aa; char* p; } obj, *ptr = &obj;  // Not anon!
```

AnyString

- Let's put all this to use
- We'll make a type that can hold any of
  - nothing
  - `std::string`
  - `std::u16string`
  - `std::u32string`

AnyString Anonymous Union

```
class AnyString {
    typedef std::string    string;
    typedef std::u16string u16string;
    typedef std::u32string u32string;
public:
    enum Contains { t_none, t_s08, t_s16, t_s32 };
private:
    Contains t_;

    union {                          // Anonymous union
        string    s08_;
        u16string s16_;
        u32string s32_;
    };
```

Discriminator

Storage

AnyString Assign() pt1

```
// Private helpers.  Note placement new.
void Assign()                                          // Default
    {                                                   t_ = t_none;}
void Assign(const string& s08)                         // Copy
    {new (&s08_)    string(s08);                        t_ = t_s08; }
void Assign(string&& s08)                              // Move
    {new (&s08_)    string(std::move(s08)); t_ = t_s08; }
void Assign(const u16string& s16)                      // Copy
    {new (&s16_) u16string(s16);                        t_ = t_s16; }
void Assign(u16string&& s16)                           // Move
    {new (&s16_) u16string(std::move(s16)); t_ = t_s16; }
void Assign(const u32string& s32)                      // Copy
    {new (&s32_) u32string(s32);                        t_ = t_s32; }
void Assign(u32string&& s32)                           // Move
    {new (&s32_) u32string(std::move(s32)); t_ = t_s32; }
```

AnyString Assign() pt 2

```
void Assign(const AnyString& anyStr)  // Copy assign
{
    switch (anyStr.t_) {
    case t_none: Assign();              break;
    case t_s08:  Assign(anyStr.s08_); break;
    case t_s16:  Assign(anyStr.s16_); break;
    case t_s32:  Assign(anyStr.s32_); break;
    }
}
```

AnyString Assign() pt 3

```
void Assign(AnyString&& anyStr)      // Move assign
{
    Contains anyStr_t = anyStr.t_;
    anyStr.t_ = t_none;
    switch (anyStr_t) {
    case t_none: Assign();                           ; break;
    case t_s08:  Assign(std::move(anyStr.s08_)); break;
    case t_s16:  Assign(std::move(anyStr.s16_)); break;
    case t_s32:  Assign(std::move(anyStr.s32_)); break;
    }
}
```

AnyString Assign() pt 4

```
// Assign from string literals uses move semantics

void Assign(const     char* cptr)
    { Assign(   string(cptr)); }

void Assign(const char16_t* cptr)
    { Assign(u16string(cptr)); }

void Assign(const char32_t* cptr)
    { Assign(u32string(cptr)); }
```

AnyString Destroy()

```
void Destroy() noexcept {
    switch (t_) {
    case t_none:                          break;
    case t_s08: s08_.~string();    break;
    case t_s16: s16_.~u16string(); break;
    case t_s32: s32_.~u32string(); break;
    }
    t_ = t_none;
};
```

AnyString Constructors

```
public:
    // Implicitly deleted constructors
    AnyString()
      { Assign();                          }
    AnyString(const AnyString& str)
      { Assign(str);                       }
    AnyString(AnyString&& str)
      { Assign(std::move(str));            }

    // Perfect forwarding constructor
    template <typename T> AnyString(T&& str)
        { Assign(std::forward<T>(str)); }
```

AnyString Assignment pt 1

```
// Implicitly deleted assignment from const reference
AnyString& operator=(const AnyString& str)
{
    if (this != &str) {  // Assignment to self?
        Destroy();       // Explicit destroy
        Assign(str);     // Placement new
    }
    return *this;
}
```

AnyString Assignment pt 2

```
// Perfect forwarding assignment
template <typename T>
AnyString& operator=(T&& str)
{
    if (this !=
        reinterpret_cast<volatile const void*>(&str))
    {
        Destroy();
        Assign(std::forward<T>(str));
    }
    return *this;
}
```

AnyString Finished!

```
    ~AnyString() noexcept
    {
        Destroy();
    }
} // Whew!
```

Consider Placement New

- `new` aligns pessimistically,
- But auto or static data may not.

```
class Doover {                          // Placement new Doover
    long double hardToAlign_;
public: Dooit() { ... }
};

static unsigned char badDooverPlace[sizeof (Doover)];
Doover* const d_ptr = new (&badDooverPlace) Doover{...};
```

- `Doover` may not be correctly aligned.

Unions and Placement New

- C++ does worst case alignment for unions.

```
class Doover {
    long double hardToAlign_;
public: Dooit() { ... }
};
static union {                  // Aligned storage for Doover
    Doover d;
};
new (&d) Doover {...};          // Placement new Doover
d.Dooit();                      // No pointer dereference!
```

- The `Doover` will be correctly aligned.

Unions Summary

- Always know what you put in your unions.
- Unions can now hold non-POD types. They require placement new and explicit destruction.
- Become aware of anonymous unions.
- Consider unions for placement new.

Enums

Executive summary:

- Enums can declare their storage type (thanks to Miller, Sutter, & Stroustrup).

- Enums with known storage type can be forward declared (thanks to Alberto Ganesh Barbati).

- A new type of enum: enum class.

Enum Thought

- An enum is C++'s simplest way to produce a new type.

- Consider enums when writing type safe code.

New Tricks for Old Enums pt 1

- An enum can (optionally) be explicitly told its underlying type.

```
enum We1 : char { snow, ice };
```

- The underlying type may be any signed or unsigned integer type.
- Values must fit in the underlying type.

```
enum We2 : signed char { value = 128 };  // error
```

New Tricks for Old Enums pt 2

- An enum of known underlying type can be forward declared.

```
enum We3 : unsigned int;
void ConsumeEnum(We3 arg);
...
enum We3 : unsigned int { small = 0, big = 0xFFFFFFFF };
```

- The forward declaration must match the definition when the compiler sees both.

```
enum E4 : char;
enum E4 { ketchup, mustard };           // error
```

New Tricks for Old Enums pt 3

- Enumerators leak into the surrounding scope (C++98/C behavior).

```
enum We5 { onion = 0, shallot = 0 };
static_assert(onion == shallot, "if you insist");
```

- Enumerators are also available in the enum's scope (new behavior).

```
static_assert(We5::onion == We5::shallot, "same effect");
```

Problems With Old Enums

- Their enumerator names contaminate the surrounding scope.

- They convert to integral type with little provocation.

Scoped Enums

- New with C++11.
- Don’t leak their enumerators.
- Don’t spontaneously convert to int.

Scoped Enum Example

```
enum class Se6 { salt = 0, seaSalt = 0, pepper };

static_assert(salt == seaSalt, "if you insist"); // error

static_assert(Se6::salt == Se6::seaSalt, "same effect");
```

- Declared with:
  - “enum class”, or
  - “enum struct”
- Same result. But I prefer “enum class”

Unscoping Enumerators

- You can selectively export enumerators from enum class by hand if you want.

```
enum class Se7 : char { soulFood, bbq, fastFood };

// Export only soulfood and bbq, not fastFood
static constexpr Se7 soulFood = Se7::soulFood;
static constexpr Se7 bbq      = Se7::bbq;

static_assert(soulFood != bbq, "Hungry?");
static_assert(Se7::soulFood != Se7::bbq, "Famished!");
```

Integral Conversion

- Old-style enums convert to int easily.

```
enum We8 : int { spoon, fork };
int we8_i = spoon;
```

- Scoped enums must be cast.

```
enum class Se8 : int { knife, napkin };
int se8_i = static_cast<int>(Se8::knife);
```

- From integral to enum uses a cast in either case.

```
We8 we8_e = static_cast<We8>(we8_i);
assert(we8_e == spoon);
Se8 se8_e = static_cast<Se8>(se8_i);
assert(se8_e == Se8::knife);
```

Enum Conversion (or not)

```
enum       We9 { pork, beef, chicken };
enum class Se9 { corn, beans, spinach };

int weDiff = We9::chicken - We9::beef;
int seDiff = Se9::spinach - Se9::beans; // error operator-

int weSum = We9::pork + 3;
int seSum = Se9::corn + 3;              // error operator+

We9 we9 { We9::chicken };
Se9 se9 { Se9::spinach };
bool test1 = false;
if (we9) { test1 = true; }
if (se9) { test1 = true; }              // error bool cvt
```

Enum Comparison

```
enum We10 { tea, soda };
static_assert(We10::tea < We10::soda, "Huh!");
We10 we10 { We10::tea };
switch (we10) {
case We10::tea:  break;
case We10::soda: break;
}                                  // No surprises
enum class Se10 { chai, coffee };
static_assert(Se10::chai < Se10::coffee, "Huh!");
Se10 se10 { Se10::chai };
switch (se10) {
case Se10::chai:   break;
case Se10::coffee: break;
}                                  // No surprises
```

std::underlying_type<>

- The `std::underlying_type<>` trait gives the data type hiding under the enum.

```
enum We11 : unsigned short { tofu, tempeh };
using We11_ut = std::underlying_type<We11>::type;
static_assert
(std::is_same<We11_ut, unsigned short>::value, "Oh!");

enum class Se11 : long long { oats, corn };
using Se11_ut = std::underlying_type<Se11>::type;
static_assert
(std::is_same<Se11_ut, long long>::value, "Surprise!");
```

Thoughts On Enum Class

- Aren't they harder to use than old-style?
- Yes. And `class` is harder to use than C-style `struct`. Type safety has a cost in convenience.
- The trick is to explore the type.

Overload Functions on Enums

Enums don't support member functions. But consider...

```
enum class Spices { vanilla, basil, pepper, curry };

bool IsHot(Spices spice)
{
    return (spice >= Spices::pepper);
}

assert (IsHot(Spices::curry));
```

Opt-In Relaxation of Enums

- Suppose we could selectively make some enums easier to convert to integral

```
enum class Se11 : char { milk, cream };

enum class Se12 : char { yoghurt, sourCream };
template <> struct enum_to_int_trait<Se12>
    : public do_enum_to_int { };

void test1() {
    auto se11int = to_integral(Se11::cream);   // error
    auto se12int = to_integral(Se12::yoghurt); // works
    assert(se12int == 0);
}
```

to_integral Implementation

```
template <typename E>
struct enum_to_int_trait
    { static constexpr bool converts_to_int = false; };
struct do_enum_to_int
    { static constexpr bool converts_to_int = true; };

// Opt-in conversion to_integral
template<typename E>
auto to_integral(E e) -> typename
std::enable_if<enum_to_int_trait<E>::converts_to_int,
typename std::underlying_type<E>::type>::type
{
    return static_cast<
        typename std::underlying_type<E>::type>(e);
}
```

Enum Iterators?

- An enum is a container of named values.
- Could we make a general iterator?
- Not until C++ has compile-time reflection
- But we could opt in specific special enums

Enum Iterator

```
enum Se13 { a = 0, min_value = a, b, c, max_value = c };

// opt-in enum iterator support
template<> struct enum_traits<Se13> :
    public enum_traits_base<
        Se13, true, contiguous_enum_tag> { };

void test2() {
    using enum_ul_t  = std::underlying_type<Se13>::type;
    enum_ul_t value = to_integral(Se13::min_value);
    for (auto ei = begin<Se13>(); ei != end<Se13>(); ++ei)
        { assert(value++ == to_integral(*ei)); }
}
```

Enums Summary

- Enums can declare their storage type (thanks to Miller, Sutter, & Stroustrup).
- Enums with known storage type can be forward declared (thanks to Alberto Ganesh Barbati).
- A new type of enum: enum class. These have better type safety than old enums.

What We Covered

- Brace Initialization
- Unions
- Enums

Sources

- Scott Meyers, *Overview of the New C++* December 12th 2012
- Pete Becker et. al., *Working Draft, Standard for Programming Language C++* February 28th 2011
- Miller, Sutter, and Stroustrup, *N2347: Strongly Typed Enums (revision 3)* July 19th 2007
- Alberto Ganesh Barbati, *N2764: Forward declaration of enumerations (rev. 3)* September 18th 2008
- Nawaz at *http://stackoverflow.com/questions/14589417/can-an-enum-class-be-converted-to-the-underlying-type* suggested the to_integral template.

- Also, many thanks to Rob Stewart for help with move semantics.

All mistakes and errors belong to Scott Schurr

Questions?

Thanks for attending