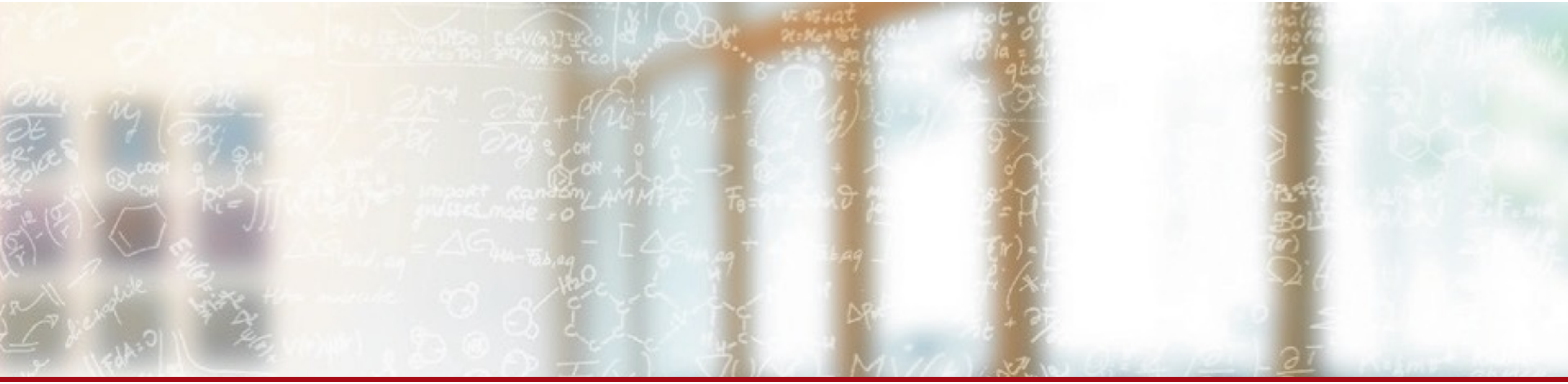
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Object Types and Value Initializations

C++ for HPC course

Mauro Bianco, CSCS

Types

- C++ is “strong typed”
  - What does it mean?
- In some languages you can write

```
x = “hello”

x = 10
```

  - What is `x`?
- In C++ after a symbol is defined it has a fixed type
  - Type can be deduced during declaration
    - Using `auto`
  - Until it goes out of scope

Type Classification

- Fundamental
  - Arithmetic
    - int, char, bool, float,…
  - void, nullptr_t
- Compound
  - References (lvalue, rvalue)
  - Pointers
  - Pointers to members
  - Array
  - Functions
  - Enumerations
  - Classes and unions
- Almost all can be cv-qualified
  - Putting `const` and `volatile` keywords

Type of Types

- Object
  - Everything is an object except references, functions and `void`
- Scalar
  - Arithmetic types, pointers, enumerations
- Trivial
  - Scalars, POD, aggregates, literals
- Incomplete
  - Useful in error messages
  - void, declarations without definitions, others

Type Traits and Concepts

- All these, and more, can be checked at compile time
- `#include <type_traits>`
  - `std::is_integral<T>::value`
  - `std::is_pod<T>::value`
  - `std::is_same<T,U>::value` (`true` if T==U, otherwise `false`)
  - … More later
- Concepts describe the requirements for the types
  - Your functions and classes works with certain type of types
  - Your functions and classes model some concepts
  - Especially useful when using templates

Object construction in C++11

- Introduces the {} initialization
- Typically () and {} equivalent
  - But there are corner cases:
  - Empty {} and ()
  - Special types like aggregates
  - Initializer lists
  - Implicit conversions

```
struct A {
    A(int a, float b);
};

int main() {
    A x(42, 3.14f);
    A y{42, 3.14f};
}
```

```
struct A {
    A();
    A(int a, float b);
};

int main() {
    A x(); /* 1 */
    A y{}; /* 2 */
    A z;   /* 3 */
}
```

```
struct B {
    B(char c) {}
};

int main() {
    char c1 = 32,
    char c2 = 33;
    B x(c1+c2); // ok
    B y{c1+c2}; // error
}
```

Value initialization

- Objects need to be initialized and this is tricky
- Different initializations
  - **Default**
    - No initializers
    - Classes calls default constructors
    - Fundamentals are indeterminate
  - **Constant**
    - Only >= C++14
    - Static or thread local that can be computed at compile time

```
int main() {
    std::vector<int> v;
    int y;
}
```

Value initialization

- Different initializations
  - **Value**
    - Empty initializers: either `()` or `{}`
    - Except:
    - T is aggregate
    - A `initiliezer_list` constructor exists

  - **Zero** (Special case of value initialization)
    - Static variables
    - Non-class value initialization : `int z{};`
    - Class data members without constructors
    - `char` arrays for elements after the end of the literal
      - `char a[10] = “hello”;`

```
int main() {
    int y{};
}
```

PODs and Aggregates

- A POD is a class
  - All data-member are POD
  - No members are references, no virtuals
  - No user defined constructors or member initializers
  - Same access control for all data members
  - An array of PODs
- Aggregate
  - All members public
  - No constructors, no virtuals, no non-public bases
  - An array type
- Aggregates can be initialized in a special way

```
class POD {
  int a;
  float b;
~~public:~~
  char c;
};
```

```
class AGG {
public:
  int a;
  float b;
  char c;
};
```

Aggregate Initialization

```
struct AGG {
  int a;  float b; char c;
};

A x = {3, 2.f, ‘c’};
```

- Also known as struct-initialization
- Uses {} to initialize data-members in order of declaration
- If a member is an aggregate {} are recursively used
- Can be done at compile time!

```
struct short_array_3 {
    int data3[3]; };
struct short_array_5 {
    int data5[5]; };
struct two_arrays {
    short_array_3 _3;
    short_array_5 _5;
    int x; };

constexpr two_arrays two = {{3,2,1},{5,4,3,2,1}, 324};
static_assert(two.x == 324, "");
```

Aggregates in real life

- std::array is an aggregate
  - Can be constructed at compile time
  - E.g., it can be used to store dimensional information
    - Compile-time loop bounds

```
constexpr std::array<unsigned, 3> bounds{{3,4,5}};
static_assert(bounds[1] == 4, "");
```

- Braces elision

```
std::array<unsigned, 3> bounds{{3,4,5}}; /* C++11 */
std::array<unsigned, 3> bounds = {3,4,5};
```

Initializer list

- std::initializer_list<T> allows the {} initialization
- More details in the examples if interested

```
template <typename T>
void f(T const& chars) {
    std::for_each(chars.begin(), chars.end(),
                  [](char c) {std::cout << c;});
    std::cout << std::endl;
}

class X {
    std::vector<char> v;
    X(std::initializer_list<char> const& chars)
        : v{chars} {
        f(chars);
    }
};
int main() {
    X x{'h', 'e', 'l', 'l', 'o'};
    X y = {'h', 'e', 'l', 'l', 'o'};

    //f({'h', 'e', 'l', 'l', 'o'});
    f(std::initializer_list<char>{'h', 'e', 'l', 'l', 'o'});
}
```

Initializer list constructor

It can be used, too

std::vector has an initializer list constructor

No deduction. Must be explicit

When use () and when {}?

- std::vector<int> v(10,4);
- std::vector<int> v{10,4};

{4,4,4,4,4,4,4,4,4,4}

{10,4}

- roundD Definition vs. Curly Construction (DD vs. CC)
- {…} for aggregates
- {…} or (...) will look for the appropriate construnctor
- But an initializer_list will take precedence!

Copy initialization

- Type x = /expression/
  - Conversion constructor selected by overload resolution
- Passing an argument by value to a function
- Returning by value

```
struct B {
    int a;
    B(int x) : a(x) {}
};

void f(B x) {}

int main() {
    B b = 42;
    f(42);
};
```

```
struct B {
    int a;
    explicit B(int x) : a(x) {}
};

void f(B x) {}

int main() {
    B b = 40; // ERROR:
    B c = static_cast<B>(10); // OK
    f(static_cast<B>(10));
};
```

- Copy elision may happens

The explicit keyword

```
struct B {
    int a;
    int b = 10;
    B(int x) : a(x) {}
    B(int x, int y) : a(x), b(y) {}
    operator int() {return a;}
};

B make_B(int a, int b) {
    return {a,b};
}

void testB() {
    B x = 10.;

    B y = {34,45};

    int i = y;
}
```

```
struct A {
    int a;
    int b = 10;
    explicit A(int x) :a(x) {}
    explicit A(int x, int y) :a(x), b(y) {}
    explicit operator int() {return a;}
};

A make_A(int a, int b) {
    return A{a,b};
}

void testA() {
    A x(10.);

    A y{34,45};

    int i = static_cast<int>(y);
}
```

Classes: default and deleted constructors

- Default constructors
  - The compiler provides implementations
    - Basically what you'd expect for PODs
- Deleted constructors
  - The compiler signal an error when invoked
    - Anyway, a deleted constructor is there

```
struct Y {
    int a = 42;
    Y() = default;
    Y(int x) {}
};

int main() {
    Y u{};
    Y w(u);
}
```

```
struct X {
    int a = 42;
    Y() = default;

};

int

    X y(x);
}
```

```
error: use of deleted function 'constexpr X::X(const X&)'
     X y(x);
         ^
note: 'constexpr X::X(const X&)' is implicitly declared as deleted
because 'X' declares a move constructor or move assignment operator
 struct X {
```

- Delete applies to any function of member function!

Automatic generation of special member functions

- Default constructor if no other constructor is explicitly declared
- Copy constructor if no move constructor and move assignment operator are explicitly declared
- If a destructor is declared generation of a copy constructor is deprecated
- Move constructor if no copy constructor, copy assignment operator, move assignment operator and destructor are explicitly declared
- Copy assignment operator if no move constructor and move assignment operator are explicitly declared
- If a destructor is declared generation of a copy assignment operator is deprecated
- Move assignment operator if no copy constructor, copy assignment operator, move constructor and destructor are explicitly declared
- Destructor

Classes: default and deleted constructors

- Default constructors
  - The compiler provides implementations
    - Basically what you'd expect for PODs
- Deleted constructors
  - The compiler signal an error when invoked
    - Anyway, a deleted constructor is there

```
struct Y {
    int a = 42;
    Y() = default;
    Y(int x) {}
};

int main() {
    Y u{};
    Y w(u);
}
```

```
struct X {
    int a = 42;
    X() = default;

    X& operator=(X&& other) = delete;
};

int main() {
    X x{};

    X y(x);
}
```

```
struct X {
    int a = 42;
    X() = default;
    X(X const& other) = default;
    X& operator=(X&& other) = delete;
};

int main() {
    X x{};

    X y(x);
}
```

- Delete applies to any member function!

Copy-Initialization Can Move

- Overload resolution happens

```
struct T {
  T() = default;
  T(T const& ) = default;
  T(T&&, int =10) {}
};

void f(T x) {}

int main() {
  T y;
  f(y);
  f(T{});
  f(std::move(y));
}
```

When doing x(y), x is copy-constructed

When doing x(T{}), x is move-constructed

x is move-constructed

Bitfields

- Useful to save space and encode information

```
struct instr_t {
    using basic_type = unsigned;
    basic_type oper  : 3;
    basic_type cond1 : 2;
    basic_type val1  : 3;
    basic_type cond2 : 2;
    basic_type val2  : 3;
    basic_type act   : 3;
    basic_type cell  : 4;
    basic_type : sizeof(basic_type)*8-21;
    basic_type :0;
    basic_type other;
};
```

- What is the size?
- Cannot be member initialized
- No overflow errors
- Compiler warning in initialization if overflown

Data alignment

- alignof(type)
  - alignof(std::max_align_t) (maximum alignment for a scalar)
- alignas
  - alignas(8) (8 bytes)
  - alignas(double) (equivalent to alignas(alignof(double)))
  - alignas(T…) (

```
int main() {
    SHOW(alignof(char));
    SHOW(alignof(int16_t));
    SHOW(alignof(int));
    SHOW(alignof(float))
    SHOW(alignof(double));
    SHOW(alignof(bool));
}
```

```
struct A {};
struct alignas(double) B {};

template <typename ...Ts>
struct alignas(Ts...) C{};

int main() {
    SHOW(alignof(A));
    SHOW(alignof(B));
    SHOW(alignof(C<char, double>));
}
```

Data alignment

- Data members can be aligned
- Affects classes sizes

```cpp
struct X {
    char a; int b; char c;
};

struct Y {
    char a; alignas(double) int b; char c;
};

int main() {
    SHOW(sizeof(X));
    SHOW(sizeof(Y));
}
```

0	a	0	a
1	pad	1	pad
2	pad	2	pad
3	pad	3	pad
4	b	4	pad
5	b	5	pad
6	b	6	pad
7	b	7	pad
8	c	8	b
9	pad	9	b
10	pad	10	b
11	pad	11	b
12		12	c
13		13	pad
14		14	pad
15		15	pad
16		16	
17		17	
18		18	
19		19	
20		20	

Data Alignment

- The alignment is for variables addresses!
  - If p is an aligned integer, then &p is aligned
  - If p is an aligned pointer, then &p is aligned
- A new(T) statement provides alignment of T

```
int main() {
    alignas(128) char cache_line[128];
    alignas(128) char* p;
    SHOW_BOOL(check_align(cache_line, 128));
    SHOW_BOOL(check_align(&p, 128));
    char p1 = '0';
    p = &p1;
    SHOW_BOOL(check_align(&p, 128));
    SHOW_BOOL(check_align(p, 128));
    SHOW_BOOL(check_align(&p1, 128));
}
```