Memory Layout, Type Deduction and Initialization 232

Part A. Memory layout
Layout refers to how various members of an object are arranged in memory.

A1. Standard layout, Trivial, POD and Aggregate

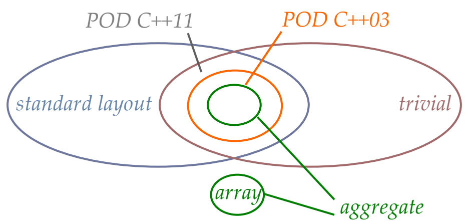
1 When a class has `vtable` compiler is free to choose its own layout, which may result in non-contiguous memory, when a class does not have `vtable`, then layout is contigous (i.e. serialization of members perhaps with ***paddings and reordering***), it is `memcpy` copyable, and `memcmp` comparable, which is fast and necessary for constructing atomic class. `memcpy` copyable class allows …

```
#define N sizeof(T)
char buffer[N];
T object0; memcpy(buffer, &object0, N); // destination, source and size
T object1; memcpy(&object1, buffer, N);
assert(object0 == object1);
```

2 The `memcpy` copyable class are then classified into several catergories :

- standard layout is *consumable* by tradition C language (as it has the same layout as C)
- trivial type is *trivially-constructable*
- aggregate is *aggregate-initializable*

3



4

	std layout	trivial	POD (C++11)
base class and members	recursive	recursive	recursive
inheritance with *no virtual member*	yes	yes	yes
members in same *inheritance layer* and *access privilege*	yes	no	yes
no non-trivial constructor / copy constructor / destructor	no	yes	yes
contiguous layout, allow `memcpy`	yes	yes	yes
no padding, no reordering, consumed by C	yes	no	yes
trivially constructable	no	yes	yes
bracket initializable `{{x,y},z}`	no	no	no

	aggregate	POD (C++03)
base class and members	recursive	recursive
no inheritance	yes	yes
members in same *access privilege*	yes	yes
no non-trivial constructor / copy constructor / destructor	yes	yes
other features	all arrays are aggregate	-
contiguous layout, allow `memcpy`	yes	yes
no padding, no reordering, consumed by C	yes	yes
trivially constructable	yes	yes
bracket initializable `{{x,y},z}`	yes	no

5 *Standard layout*

Standard layout requires all members having same access privilege and in same inheritance layer, then its layout is the same as that in traditional C (there is no padding nor reordering of members). Its pointer can be `reinterpret_cast` into pointer to first member.

```
T* pT = new T;
A* pA = reinterpret_cast<A*>(pT);              // suppose A = typeof(T::member1)
```

6 *Trivial type*

Trivial type refers to classes that can be constructed trivially, through compiler-generated constructor. As trivial type has members from different layers of inheritance tree, or having different access privileges, its layout may have padding bytes between members or reordering of members, hence trivial cannot be consumed by other languages. There are 3 approaches to generate trivial default constructor or copy/move constructor :

1. do not declare any constructor, compiler will generate an implicit version like `class T0`
2. declare constructor with `=default` like `class T1`
3. define constructor with `=default` like `class T2` (but this method does not result in a trivial class)

```
class T0 { };                               // T0 is     trivial and     movable.
class T1 { T1() = default; };               // T1 is     trivial and non-movable.
class T2 { T2(); };  T2::T2() = default;    // T2 is non trivial and non-movable.
```

	Is trivial class?	Is move implicit?	Control access privilege?
do not declare constructor	yes	yes	no
declare constructor `=default`	yes	no	yes
define constructor `=default`	no	no *compare to rvalue.doc C7*	yes

7 *POD in C++11 and POD in C++03*

1. Under C++11 specification, plain old data is defined as intersection between standard layout and trivial.
2. Under C++03 specification, plain old data is stricter, no inheritance is allowed.

- In `YLib`, we restrict `event` class from having custom constructor, so that `event` is POD, easy for serialization

8 *Aggregate*

Aggregate has three parts :

1. Firstly it is a stricter POD, there is no inheritance, all members are public.
2. Secondly all arrays are aggregate, array of non-aggregate type is also aggregate.
3. Finally, default-member-initializer makes a `struct` **non-aggregate**.

The major characteristics of aggregate is *aggregate initialization*, which is a memberwise-initialization with brace. If number of initial values is less than the number of members in the aggregate, *value initialization* is triggered for the rest (read latter section for details), if number of initial values is too many, it will result in compile errors. Aggregate initialization is possible if all members are public.

```
    struct T { int a,b; int c[3]; };
    struct S { int x;    T y;      void f(){} };

    S s1 = {1,{2,3,{4,5,6}}}; // aggregate-initialization
    S s2 = {1, 2,3, 4,5,6  }; // aggregate-initialization with brace-elision
    S s3   {1,{2,3,{4,5,6}}}; // aggregate-initialization with direct-list-initialization syntax
    S s4   {1, 2,3, 4,5,6  }; // aggregate-initialization with direct-list-initialization syntax and brace-elision (C++14)
//  S s5 = {1, 2,{3,4,5,6} }; // compile error
```

More about array initialization :

```
    int array2d_1[2][2] = {{1,2},{3,4}};
    int array2d_2[2][2] = { 1,2,  3,4 };       // with brace elision
    int array2d_3[2][2] = {{1  },{3  }};       // equivalent to {{1,0},{3,0}}, value-initialization of int is zero

    T arr0[4] = { T{a,b,c},T{},T{x,y},T{z} };
//  T arr1[4] = arr0;                          // compile error, we cannot initialize like that, instead, we should ...
    T arr1[4];  memcpy(arr1, arr0, sizeof(T)*4);

    // similarly, for char array, we have :
    T ac0[4] = { 'a','b','c','\n' };           // or equivalently ...
    T ac0[4] = "abc";
//  T ac1[4] = ac0;                            // compile error, we cannot initialize like that, instead
//  T ac1[4] = std::string{"abc"}.c_str();     // compile error, we cannot initialize like that, instead
```

Here is an experiment.

```
struct B
{
};

struct B3
{
    std::uint8_t x;
    std::uint8_t y;
    std::uint8_t z;
};

struct BV
{
    virtual void f() = 0;
};

// *** Non contiguous *** //
struct D0 : public BV
{
};

// *** Contiguous, non-std-layout, non-trivial *** //
struct D1 : public B3
{
    std::uint8_t u = 1;
};

struct D2 : public B
{
    D2(){}
    public:  std::uint8_t u;
    private: std::uint8_t v;
};

// *** Contiguous, non-std-layout, trivial *** //
struct D3 : public B3
{
    std::uint8_t u;
};

struct D4 : public B
{
    D4() = default;
    public:  std::uint8_t u;
    private: std::uint8_t v;
};

// *** Contiguous, std-layout, trivial *** //
struct D5 : public B3
{
    void fct();
};

struct D6 : public B
{
    D6() = default;
    public:  std::uint8_t u;
             std::uint8_t v;
};


std::cout << std::is_standard_layout<D0>::value << " " << std::is_trivial<D0>::value; // false false
std::cout << std::is_standard_layout<D1>::value << " " << std::is_trivial<D1>::value; // false false
std::cout << std::is_standard_layout<D2>::value << " " << std::is_trivial<D2>::value; // false false
std::cout << std::is_standard_layout<D3>::value << " " << std::is_trivial<D3>::value; // false true
std::cout << std::is_standard_layout<D4>::value << " " << std::is_trivial<D4>::value; // false true
std::cout << std::is_standard_layout<D5>::value << " " << std::is_trivial<D5>::value; // true true
std::cout << std::is_standard_layout<D6>::value << " " << std::is_trivial<D6>::value; // true true
```

Reference Above summary is consistent with the following references :

- *Trivial, standard layout, POD and literal types*, written by docs.microsoft.com
- *What are aggregates and PODs and why are they special*, in stackoverflow.com

```
std::cout << std::boolalpha << std::is_standard_layout<T>::value;
std::cout << std::boolalpha << std::is_trivial<T>::value;
std::cout << std::boolalpha << std::is_pod<T>::value;
```

A2. Structure padding and alignment requirement

Structure padding and alignment requirement are closely related, yet different ideas.

1 What is structure padding *[JP interview]*? Zero padding are added so that size of `struct` is multiple of 32bits or 64bits depending on the *CPU* and *OS*, so that the *CPU* can access the memory faster. Consider reading 16 bytes `struct` starting from an arbitrary starting address in 64 bits machine, it involves reading one partial 8 bytes, one full 8 bytes, a final partial 8 bytes. All partial 8 bytes involves calculation of offset and opsize, which takes long time in *CPU* circuitry.

```
struct my_type
{
    char c;               // 1 byte
    int i;                // 4 bytes
    double d;             // 8 bytes
};
std::cout << sizeof(A);     // 16! Why? Zero padding in struct so as to facilitate memory access
```

2 How to disable structure padding *[Citadel interview]*? This technique is useful for applying `reinterpret_Cast` in datafeed parser.

```
#pragma pack(push, 1)
struct my_type
{
    char c;               // 1 byte
    int i;                // 4 bytes
    double d;             // 8 bytes
};
#pragma pack(pop)
std::cout << sizeof(A);     // 13
```

3 What is alignment requirement of `struct`? Alignment is a restriction on memory position on which the first byte of object is stored, alignment of 16 means memory addresses are restricted to multiples of 16 (we can only align to powers of 2, that are 1, 2, 4, 8, 16 ...). It is used mainly for two purposes :

- implementation of cache friendly container, improve cache hit, avoid false sharing etc
- usage of instructions or library that work only on data with particular alignment such as `SSE` or `AVX`

4 Two commonly used functions are `alignas` and `alignof`.

```
alignas(16)       int a[10];
alignas(16*16)    int b[10];
alignas(16*16*16) int c[10];
std::cout << "\n" << a;              // 0x7f2116e82e70, 1 zero  at the end for multiple of 16
std::cout << "\n" << b;              // 0x7f2116e82c00, 2 zeros at the end for multiple of 16*16
std::cout << "\n" << c;              // 0x7f2116e82000, 3 zeros at the end for multiple of 16*16*16
std::cout << "\n" << &(a[1]);        // 0x7f2116e82e74, not aligned
std::cout << "\n" << &(b[1]);        // 0x7f2116e82c04, not aligned
std::cout << "\n" << &(c[1]);        // 0x7f2116e82004, not aligned
```

The above aligns the first element of each array only, the second element is not aligned. To align each element, we need :

```
alignas(16)       int a0, a1, a2;
alignas(16*16)    int b0, b1, b2;
alignas(16*16*16) int c0, c1, c2;
std::cout << "\n" << &a0;            // 0x7f2116e82e70
std::cout << "\n" << &a1;            // 0x7f2116e82e80
std::cout << "\n" << &b0;            // 0x7f2116e82c00
std::cout << "\n" << &b1;            // 0x7f2116e82d00
std::cout << "\n" << &c0;            // 0x7f2116e82000
std::cout << "\n" << &c1;            // 0x7f2116e83000
```

Function `alignof` finds the alignment that we apply. This is not the same as applying modulus operator on address.

```
alignas(16)       int a[10];
alignas(16*16)    int b[10];
alignas(16*16*16) int c[10];
std::cout << "\n" << alignof(a);       // 16
std::cout << "\n" << alignof(a[1]);    // 4
std::cout << "\n" << alignof(b);       // 256
std::cout << "\n" << alignof(b[1]);    // 4
std::cout << "\n" << alignof(c);       // 4096
std::cout << "\n" << alignof(c[1]);    // 4
```

Part B. Auto deduction and declared type deduction

In C++11, both `auto` and `decltype` are compile-time type deduction. The former is deduced from initialization statement whereas the latter is deduced from an expression supplied by the caller inside a bracket.

```
    auto x = expression; // initialization statement
    decltype(expression) x;

//  typedef auto ??      // compile error
    typedef decltype(expression) my_type;
```

B1. Mechanism for `auto` and `decltype`

- `auto` deduction share the same mechanism as template deduction
- `auto` deduction differs from template deduction that `auto` can deduce `initializer_list` from brace init, while template cannot
- `auto` deduction must come along an initialization, it cannot be used in `typedef`
- `decltype` deduction depends on the complexity of bracketed expression
- rule `2.1-2.3` are identical to those for resolving universal reference in *Rvalue and move doc*

For non universal reference, such as `auto a = expression` or `auto& a = expression` or `const auto& a = expression` or …

`1.1` ***reference stripping*** of expression type (pointer adornment `*` is not stripped away)

`1.2` ***constant stripping*** of previous result, as long as modifiability is unchanged

`1.3` substitute previous result into `auto` or `auto&` or `const auto&`, perform ***reference collapsing*** if necessary

For universal reference, such as `auto&& a = expression`

`2.1` ***reference stripping*** of expression type (pointer adornment `*` is not stripped away), suppose result is `A`

`2.2` resolving universal reference to `A&` for `lvalue` expression or `A` for `rvalue` expression (*NO* `A&&`)

`2.3` substitute previous result into `auto&&=(A&)&&` or `auto&&=(A)&&`, perform ***reference collapsing*** if necessary

For `decltype` with simple expression or member access, such as `decltype(x)` or `decltype(ptr->m)`

`3.1` no reference stripping nor `const` stripping

`3.2` deduced type is type of the expression as declared

For `decltype` with complex expression or bracketed expression, such as `decltype((x))` or `decltype((ptr->m))` or `decltype(x*y)`

`4.1` no reference stripping nor `const` stripping

`4.2` if `A` is type of the expression as declared, then :

- deduced type is `A&` for `lvalue` expression
- deduced type is `A&&` for `xvalue` expression
- deduced type is `A` for `prvalue` expression

`4.3` perform ***reference collapsing*** if necessary

Examples for `auto`

Constant adornment is *NOT* stripped away if we declare `auto&` or `auto&&` pointing to constant object, otherwise we modify it.

```
struct M {      };
struct X { M m; };
X   x;
X& rx = x;          const X& rcx = x;
X* px = new X;      const X* pcx = px;

// strip & and strip const            // strip & and append &            // strip &, universal ref and append &&
auto y0 = x;        // X              auto& ly0 = x;      // X&          auto&& ry0 = x;            // X& && = X&
auto y1 = rx;       // X              auto& ly1 = rx;     // X&          auto&& ry1 = rx;           // X& && = X&
auto y2 = rcx;      // X              auto& ly2 = rcx;    // const X&    auto&& ry2 = rcx;          // const X& && = ...
auto p0 = px;       // X*                                                auto&& ry3 = std::move(x); // X  && = X&&
auto p2 = pcx;      // const X*
auto m0 = px->m;    // M              auto& lm0 = px->m;  // M&          auto&& rm0 = px->m;        // M& && = M&&
auto m2 = pcx->m;   // M              auto& lm2 = pcx->m; // const M&    auto&& rm2 = pcx->m;       // const M& && = ...

// if we want to keep &, const        // const-stripping is skipped, so we can protect against modifying x via ly2
const auto& z = rx;                   ly2 = new_value;    // compile error
```

Examples for `decltype`
Simple expression refers to variable with no parenthesis or simply member access. Complex expression refers to everything else.

```
// Thomas Becker - simple expressions
X   x;
X& rx = x;       const X& rcx;
X* px = new X;   const X* pcx = px;

auto y0 = x;      // X                        typedef decltype(x)        type;     // X
auto y1 = rx;     // X                        typedef decltype(rx)       type;     // X&
auto y2 = rcx;    // X                        typedef decltype(rcx)      type;     // const X&
auto p0 = px;     // X*                       typedef decltype(px)       type;     // X*
auto p1 = pcx;    // const X*                 typedef decltype(pcx)      type;     // const X*
auto m0 = px->m;  // M                        typedef decltype(px->m)    type;     // M
auto m2 = pcx->m; // M                        typedef decltype(pcx->m)   type;     // M (no const, verified in gcc)

// Thomas Becker - complex expressions by parenthesizing simple expressions
                                              typedef decltype((x))      type;     // X  &       = X&       (lvalue)
                                              typedef decltype((rx))     type;     // X& &       = X&       (lvalue)
                                              typedef decltype((rcx))    type;     // const X& & = const X& (lvalue)
                                              typedef decltype((px->m))  type;     // M  &       = M&       (lvalue)
                                              typedef decltype((pcx->m)) type;     // const M  & = const M& (lvalue)

// Thomas Becker - complex expressions with binary and ternary operators
int x = 1;     const int cx = 11;     double dx = 1.0;
int y = 2;     const int cy = 12;     double dy = 2.0;     const X& fct();

auto z =  x *  y;      // int                 typedef decltype( x* y) type;        // int                  (prvalue)
auto z = cx * cy;      // int                 typedef decltype(cx*cy) type;        // int                  (prvalue)
auto z = dx<dy? dx:dy; // double              typedef decltype(dx<dy? dx:dy) type; // double   & = double&  (lvalue)
auto z = cx<dy? cx:dy; // double (promoted)   typedef decltype(cx<dy? cx:dy) type; // double               (prvalue)
auto z = fct();        // X                   typedef decltype(fct()) type;        // const X& & = const X& (lvalue)
```

- as `(x*y)` and `(cx*cy)` return temporary unnamed variable, the result is `prvalue`, hence no `const` for `(cx*cy)` result
- as `(dx<dy?dx:dy)` returns either one existing variable, the result is `lvalue`
- as `(cx<dy?cx:dy)` returns a temporary promoted double, the result is `prvalue`

More examples
We test `auto` together with `decltype` using online gcc compiler.

```
#include <type_traits>
int        x;               auto&& p = x;
int&       y = x;           auto&& q = y;
const int& z = x;           auto&& r = z;
int&&      w = std::move(x); auto&& s = std::move(x);

// We repeat the following for variable q,r,s too. We omit duplicated code for clarity.
if (std::is_same<decltype(p), int>::value)                std::cout << "\ntype : int";
if (std::is_same<decltype(p), int*>::value)               std::cout << "\ntype : int*";
if (std::is_same<decltype(p), int* const>::value)         std::cout << "\ntype : int* const";
if (std::is_same<decltype(p), const int*>::value)         std::cout << "\ntype : const int*";
if (std::is_same<decltype(p), const int* const>::value)   std::cout << "\ntype : const int* const";
if (std::is_same<decltype(p), const int&>::value)         std::cout << "\ntype : const int&";
if (std::is_same<decltype(p), int&>::value)               std::cout << "\ntype : int&";
if (std::is_same<decltype(p), int&&>::value)              std::cout << "\ntype : int&&";

// Here are the answers :
decltype(x)     = int
decltype(y)     = int&
decltype(z)     = const int&
decltype(w)     = int&&

decltype((x))   = int&
decltype((y))   = int&
decltype((z))   = const int&
decltype((w))   = int&
decltype((x+2)) = int

decltype(p)     = int&
decltype(q)     = int&
decltype(r)     = const int&
decltype(s)     = int&&
```

B2. Applications for `auto`

```
// (1) specific type
auto s0 = "abcdef";           // deduced as char*
auto s1 = "abcdef"s;          // deduced as std::string
auto i0 = 123;                // deduced as int
auto i1 = 123ul;              // deduced as unsigned long

// (2) for-loop iterating through container
const std::vector<X> cvec;
      std::vector<X>  vec;
for(auto i=cvec.begin(); i!=cvec.end(); ++i) { i->print();  }
for(auto i= vec.begin(); i!= vec.end(); ++i) { i->modify(); }
for(auto& x:cvec) { x.print();  }  // handle both cvec and vec with "auto&"
for(auto& x: vec) { x.update(); }  // this is why auto does not trim const when constness is violated

// (3) ordinary function input (known as Abbreviated function template, available in c++20)
//       lambda function input
void fct(const auto& arg0, auto&& arg1, my_concept auto arg2); // please read concepts too
std::for_each(vec.begin(), vec.end(), [](auto& x){ std::cout << x; });

// (4) template function local variable
template<typename T, typename S>
void fct(const T& lhs, const S& rhs) { auto intermediate = lhs * rhs; ... }

// (5) ordinary function return
//     template function return
//       lambda function return (no auto keyword needed)
template<typename T, typename S>
auto fct(const T&      lhs, const S&      rhs) { return lhs * rhs; }
auto fct(const matrix& lhs, const vector& rhs) { return lhs * rhs; }
      [](const matrix& lhs, const vector& rhs) { return lhs * rhs; }
```

Before C++14, TRTS is needed, that is, adding `->decltype(...)`

```
// - for multiple returns, they should return consistent type
// - for with recursion call, boundary case must be sequenced first
auto fct() // multi-returns
{
    if (condition) return 0;
    else return 1;
}
auto fct(int n) // recursion
{
   if (n <=2) return 1;
   else return fct(n-2)+fct(n-1);
}

// (6) lambda function wrapper
auto fct0 = [](int x){ std::cout << x; };
auto fct1 = std::bind(fct, x, y);
```

B3. Applications for `decltype`

- `decltype` is useful for `typedef` in which no new variable is declared, where `auto` is inapplicable
- `decltype` and `auto` appear together in *Trailing Return Type Syntax* TRTS, yet it is `decltype` that does the deduction

```
// (1) define typedef
typedef decltype(vec.begin()) iterator_type;

// (2) decltype is compile time deduction, expression is not run, access out of bound is OK
typedef decltype(vec[100]) reference_type;

// (3) decltype can access nested type
typedef decltype(vec)::value_type value_type;

// (4) reference can be removed by
typedef std::remove_reference<decltype(vec[vec.size()])>::type value_type;
```

Iterator-type, reference-type and value-type respectively

```
// (5) forward universal reference in Abbreviated function template
void fct(const auto& arg0, auto&& arg1, my_concept auto arg2)
{
    impl(std::forward<decltype(arg1)>(arg1)); // arg1 is universal reference
}

// (6) before C++14, return type can be deduced only with TRTS
template<typename T, typename S>
auto fct(const T& lhs, const S& rhs) -> decltype(lhs * rhs) { return lhs * rhs; }

// (7) after C++14, we can replace clumbersome decltype in the 1st line by the 2nd line, they are equivalent
decltype(clumbersome_expression) x = clumbersome_expression;  // C++11
decltype(auto) x = expression;                                // C++14

// (8) decltype can be used as base class
class my_array : public decltype(vec)
{
    public:  my_array() : decltype(vec)(1024) {}
}
```

base class *initialization of base class*

Part C. Initialization is bonkers

Here are various C++ initializations. According to the context, we put them into different catergories. Please note :

- different initializations are not mutually exclusive, they may be intersecting sets,
- C++ committee is struggling with choice of words, beware of difference in terminologies,
  - initializer list ≠ list initialization
  - initializer refers to class member
  - initialization refers to other cases
  - there is no ~~brace-or-equal initialization~~
- for each initialization, please find out (1) <u>how it is triggered</u> and (2) <u>what it does</u>.

For initializing local variable

1	zero-initialization	
2	default-initialization	no bracket, no init value
3	value-initialization	with bracket but no init value
4	direct-initialization	with bracket and value
5	copy-initialization	equals to bracketed value
6	list-initialization	brace counterpart of the two rows above
7	`std::initializer_list<>`	
8	aggregate-initialization	
9	uniform-initialization	

For initializing class member

- member-initializer-list
- default-member-initializer (C++ committee picked this name, according to Richard Smith in 2015)
- = brace-or-equal initializer
- designated initializer

C1. Initializing local variable

Here is the summary showing how different initializations are triggered (mostly copied from Christian Aichinger's thoughts, some items are omitted on purpose, for the sake of clarity). By comparing bracket-syntax with brace-syntax, for the entries inside red box, we can find that the brace-syntax is more complete, there are missing entries for bracket syntax, which are crossed out because they cause ambiguity between constructor invocation and function declaration. That is why brace initialization syntax is introduced.

`T t(); declares function rather than variable`

		named obj	unnamed obj	mem init list	default mem init
2	default init	`T t;`	-	-	`class S { T m; };`
3	value init	-	`fct(T());`	`S::S():m() {}`	-
4	direct init	`T t(x,y,z);`	`fct(T(x,y,z));`	`S::S():m(x,y,z) {}`	-
5	copy init	`T t = t0;`	-	-	-
6.3	value-list init	`T t{};`	`fct(T{});`	`S::S():m{} {}`	`class S { T m{}; };`
6.4	direct-list init	`T t{x,y,z};`	`fct(T{x,y,z});` `fct({x,y,z});`	`S::S():m{x,y,z} {}`	`class S { T m{x,y,z}; };`
6.5	copy-list init	`T t = {x,y,z};`	-	-	`class S { T m = {x,y,z}; };`

A very useful technique in `YLib`*, which is used to zero-set all POD members in a class.*

```
struct T
{
    T() : x(0), y(1) {}
    T(int xx, int yy) : x(xx), y(yy) {}
    void fct() { cout << x << y; }

    int x,y;
};
void fct(const T& t) { cout << t.x << t.y; }

    fct(T());          //  0,  1             T().fct();         //  0,  1
    fct(T(10,11));     // 10, 11             T(50,51).fct();    // 50, 51
//  fct((20,21));      // compile error
    fct(T{});          //  0,  1             T{}.fct();         //  0,  1
    fct(T{30,31});     // 30, 31             T{60,61}.fct();    // 60, 61
    fct({40,41});      // 40, 41
```

Zero / default and value-initialization

Zero / default and value initialization are initializations without user-explicitly-specified values. Zero initialization, and sometimes, even default initialization, are triggered as a part of value initialization. Besides default initialization can also be triggered explicitly by instantiation without bracket while value initialization is triggered by instantiation with empty bracket. The following is a list of what they do, which is consistent with reference *"Initialization in C++ is bonkers"* and *"Brace, brace!"*.

zero-initialization

if `T` is a scalar type (i.e. `enum`, `int`, `double`, pointer)	set to zero
if `T` is a class	each member is zero-initialized
if `T` is an array	each element is zero-initialized

default-initialization

if `T` is a scalar type (i.e. `enum`, `int`, `double`, pointer)	invoke nothing
if `T` is a class	invoke default constructor *(user-defined or compiler-generated)*
if `T` is an array	each element is default-initialized

value-initialization

if `T` is a scalar type (i.e. `enum`, `int`, `double`, pointer)	the object is zero-initialized
if `T` is a class	
if `T` is trivial	the object is zero-initialized and then default-initialized
else	the object is default-initialized
if `T` is an array	each element is value-initialized

summarise in table form	*scalar-type*	*class*	*array*
zero-initialization	*set zero*	*zero-initialization*	*zero-initialization*
default-initialization	*nothing*	*default-constructor*	*default-initialization*
value-initialization	*zero-initialization*	*depends*	*value-initialization*

Remarks

- by placing `=default` in different locations, result in differernt initializations for `T0/T1/T2`

```
class T0 { };                               // There is compiler-generated constructor. T0 is trivial, goto
class T1 { T1() = default; };               // This  is compiler-generated constructor. T1 is trivial, goto
class T2 { T2(); };  T2::T2() = default;    // This  is user-defined constructor.       T2 is NOT trivial, goto
```

- class members omitted in member-initializer-list is default-initialized
- class members followed by empty bracket in member-initializer-list is value-initialized

```
class U
{
    U():m1() {}                             // m0 is default-initialized
    T m0,m1;                                // m1 is value-initialized
};
```

Direct and copy-initialization

Unlike the previous initializations direct and copy initializations are done with user-explicitly-specified values. Direct-initialization does not have explicit assignment `=`, while copy-initialization does. Besides the latter is triggered when objects are implicitly copied, like during *pass-by-value*, *return-by-value* and *catch-by-value*. Difference between direct and copy initializations is what they do :

direct initialization

resolution among all `explicit` constructors and all `non-explicit` constructors

copy initialization

resolution among all `non-explicit` constructors and all conversion operators

List-initialization *(read reference Brace, brace!)*

We introduce brace syntax to solve the ambiguity problem caused by bracket syntax. All initializations with brace syntax are called list-initialization. Depending on the class type, it forwards the list-initialization to either :

1. if class `T` is constructed with bracketed arguments, then ...
- resolves the best among all constructors
- **cannot** invoke aggregate initialization
2. if class `T` is constructed with braced arguments, and ...
- if class `T` is an aggregate, then forward braced arguments to aggregate initialization
- otherwise if class `T` offers constructor taking `std::initializer_list<U>`, then forward braced arguments to that constructor
- otherwise resolves the best among all constructors, invoking value-list / direct-list / copy-list-initialization
  value-list / direct-list / copy-list-initialization are counterparts of value / direct / copy-initialization respectively
3. if class `T` has constructors taking non `std::initializer_list<U>` and taking `std::initializer_list<U>` like `std::vector`
- how to invoke the former? use bracket `std::vector<int> v0(10, 123)` to construct a vector of ten elements, all value 123
- how to invoke the latter? use brace `std::vector<int> v1{10, 123}` to construct a vector of two elements, value 10 and 123

```
struct rgb
{
    rgb() = default;
    rgb(int r_, int g_, int b_) : r(r_),g(g_),b(b_) {} // User-define constructor to make it non-aggregate
    int r,g,b;
};

struct pixel
{
    pixel() = default;
    pixel(int x_, int y_, int r_, int g_, int b_) : x(x_),y(y_),c(r_,g_,b_) {} // User-define constructor
    int x,y; rgb c;
};

void fct(const pixel& p);

// Named instance
pixel p0(100, 120, 0xff, 0xff, 0x00);
pixel p1{100, 120, 0xff, 0xff, 0x00};
pixel p2 = {100, 120, 0xff, 0xff, 0x00};
pixel p3; p3 = {100, 120, 0xff, 0xff, 0x00};

// Unnamed instance
fct(pixel(100, 120, 0xff, 0x00, 0xff));
fct(pixel{100, 120, 0xff, 0x00, 0xff});
fct({100, 120, 0xff, 0x00, 0xff});

T factory() { return {100, 120, 0xff, 0xff, 0x00}; }

// Array initialization (which cannot be achieved by bracket)
rgb  colors[4]    {{r0,g0,b0}, {r1,g1,b1}, {r2,g2,b2}, {r3,g3,b3}};
rgb* p=new rgb[4] {{r0,g0,b0}, {r1,g1,b1}, {r2,g2,b2}, {r3,g3,b3}};
```

*Brace syntax allows **omitting class name** :*
(1) in construction
(2) in function argument
(3) in function return

Template class `std::initializer_list<T>`

1. Array is a fixed size `std::vector<T>`.
2. `std::initializer_list<T>` is a constant array of homogenous elements.
3. `std::initializer_list<T>` must be constructed with brace-syntax. Using bracket syntax will result in compile error.

```
//  std::initializer_list<int> a0 (1,2,3,4,5); for(const auto& x:a0) std::cout << x << " "; // compile error
//  std::initializer_list<int> a1=(1,2,3,4,5); for(const auto& x:a1) std::cout << x << " "; // compile error
    std::initializer_list<int> a2 {1,2,3,4,5}; for(const auto& x:a2) std::cout << x << " ";
    std::initializer_list<int> a3={1,2,3,4,5}; for(const auto& x:a3) std::cout << x << " ";
```

Initializer list is a constant array (a degraded `std::vector<T>`) which offers limited functions :

```
typedef typename std::initializer_list<T>::iterator iter_type;
iter_type std::initializer_list<T>::begin() const;
iter_type std::initializer_list<T>::end() const;
size_t std::initializer_list<T>::size() const;
const T& std::initializer_list<T>::operator[](int n) const;
```

In order to support list-initialization of class `T` from arguments of type `U`, we have to implement :

```
T::T(const std::initializer_list<U>& init)
{
    for(const auto& x:init) do_something(x);
}
T t0{1,2,3,4,5};
T t1 = {1,2,3,4,5,6,7,8,9};
```

Aggregate-initialization

For aggregate types, brace-syntax initializaton `{}` becomes an aggregate-initialization :

1. it is a memberwise initialization, members are initialized in the order of their declarations
2. if number of braced elements is less than class members, extra members are value-initialized
3. if number of braced elements is more than class members, there will be compile error.

- as aggregate has no user-defined constructor nor default-member-initializer, members are all value-initialized
- value initialization for scalar type is zero-setting done in compile-time (a common technique in YLib)
- aggregate is thus useful for building protocol for low latency datafeed :

```
struct protocol
{
    int m0[10];
    char m1[10];
    double m2[10];
};

std::ostream& operator<<(std::ostream& os, const protocol & x)
{
    os << "\nis aggregte = " << std::is_aggregate<protocol>::value;
    os << "\nm0 = "; for(int n=0; n!=10; ++n) os << (int)(x.m0[n]) << " ";
    os << "\nm1 = "; for(int n=0; n!=10; ++n) os << (int)(x.m1[n]) << " ";
    os << "\nm2 = "; for(int n=0; n!=10; ++n) os << x.m2[n] << " ";
    return os;
}
```

Please note that for aggregate types, aggregate initialization is triggered by brace-syntax. See the difference between `x0` and `x1`.

```
    protocol x0;   // invoke default initialization, i.e. random values
    protocol x1{}; // invoke aggregate initialization, which invokes value initialization, in turn, set zero for scalars
    protocol x2{1,2,3,4,5,6,7,8,9};
    protocol x3{{1,2,3,4},{5,6,7},{8,9}};
//  protocol x4{{1},{2},{3},{4},{5},{6}}; // compile error, too many arg

    x0.m0 = random numbers ...
    x0.m1 = random numbers ...
    x0.m2 = random numbers ...
    x1.m0 = 0 0 0 0 0 0 0 0 0 0
    x1.m1 = 0 0 0 0 0 0 0 0 0 0
    x1.m2 = 0 0 0 0 0 0 0 0 0 0
    x2.m0 = 1 2 3 4 5 6 7 8 9 0
    x2.m1 = 0 0 0 0 0 0 0 0 0 0
    x2.m2 = 0 0 0 0 0 0 0 0 0 0
    x3.m0 = 1 2 3 4 0 0 0 0 0 0
    x3.m1 = 5 6 7 0 0 0 0 0 0 0
    x3.m2 = 8 9 0 0 0 0 0 0 0 0
```

Uniform initialization

In C++03, initialization syntax is different for different data types, such as :

```
int n = 3;                                        // =  for integer
int m[] = {1,2,3};                                // {} for array
std::string str;                                  // {} for default-initialization
std::string str("This is a string.");             // () for direct-initialization
```

In C++11, with extended capability of brace-syntax, initialization syntax of all data types are standardized, hence the name uniform initialization. This is particularly useful for template programming, as we have a standard way to initialize `T` inside template.

```
T t{};                                            // example 1 :  value list initialization
T t{x,y,z};                                       // example 2 : direct list initialization
T t = {x,y,z};                                    // example 3 :   copy list initialization
T array[] = {A,B,C,D,E};                          // example 4 :   aggregate initialization
std::vector<int> vec{1,2,3,4};                    // example 5 : initializer list for vector
std::map<string,int> map{{"ABC",1},{"DEF",2}};    // example 5 : initializer list for map
```

C2. Member-initializer-list / Default-member-initializer

We discussed initialization of local variables, now we move forward to non-static class members, they are initialized in 2 ways :

- member-initializer-list *in constructor* or using brace-syntax or bracket-syntax (direct initialization)
- default-member-initializer *in member declaration* using brace-syntax or equal-syntax (direct or copy initialization)

Therefore, default-member-initializer is also known as brace-or-equal initializer. It invokes direct initialization for brace-syntax and copy initialization for equal-syntax. If a member is initialized using both member-initializer-list and default-member-initializer, the latter will be ignored *(not overwritten)*. Member-initializer-list and default-member-initializer are invoked prior to constructor body.

```
struct S
{
    S(int x_) : x{x_} { }        // member-initializer-list
    int x;
};
struct T
{
    T() : s{10} { }             // member-initializer-list
    S s{11};                    // default-member-initializer with direct-initialization
//  S s = S(11);                // default-member-initializer with copy-initialization
//  S s = S{11};                // default-member-initializer with copy-initialization
//  S s =  {11};                // default-member-initializer with copy-initialization
};
```

With *default-member-initializer*, the class will **no longer be an aggregate**, aggregate-initialization becomes invalid (see A1).

C2.(continue) Designated initializer

Furthermore we can initialize POD by specifying its members with the dot syntax (however members must be in order) :

```
struct S
{
    int x = 1;
    int y = 2;
    int z = 3;
};

S s{ .x = 11, .z = 33 };
std::cout << s; // 11, 2, 33
```

Reference

- Overview of C++ Variable Initialization, in Christian Aichinger's thoughts *[about how different initializations are triggered]*
- Initialization in C++ is bonkers, by Simon Brand *[about what default/value initializations do]*
- Thoughts on the Vagaries of C++ Initialization, by Scott Meyers *[about what direct/copy initializations do]*
- Brace, brace! by Andrzej Krzemienski *[about what brace syntax does]*