Constructor/Destructor

- Constructor as conversion operator
- How to define operators
- Initialization list
- Destructors / array destruction
- Cloning
- Delegated constructor
- Using constructor
- Universal constructor
- Construction and exceptions

Constructor/Destructor

- In OO languages constructors to initialize objects
- Other style: using factory objects
- Constructors are also
  - Conversion operators
  - Create temporaries
- Destructors are to free resources
  - Resource: much more than memory
  - e.g. unlock, fclose, ...

A constructor quiz

```
#include <iostream>

struct foo {
    foo() = default;
    int a;
};

struct bar {
    bar();
    int b;
};

bar::bar() = default;

int main() {
    foo a{};
    bar b{};
    std::cout << a.a << ' ' << b.b;
}
```

Default-, value- and zero-initialization

```
T global;    // zero-initialization,
             // then default-initialization

void foo() {
  T i;       // default-initialization
  T j{};     // value-initialization (C++11)
  T k = T(); // value-initialization
  T l = T{}; // value-initialization (C++11)
  T m();     // function-declaration
  new T;     // default-initialization
  new T();   // value-initialization
  new T{};   // value-initialization (C++11)
}

struct A { T t; A():t() {} }; // t is value-initialized

struct B { T t; B():t{} {} }; // t is value-initialized(C++11)

struct C { T t; C()     {} }; // t is default-initialized
```

Default-, value- and zero-initialization

- Default initialization
  - If T is a class: default constructor is called
  - If an array, each element is default initialized
  - Otherwise: no initialization, results indeterminate value
- Value initialization
  - If T is a class: default initialized
    - (after zero initialized if T's default constructor is not user defined / deleted)
  - If an array, each element is value initialized
  - Otherwise: zero initialized
- Zero initialization
  - Applied to static and thread local variables before any other initialization
  - If T scalar (number, pointer, enum) set to 0
  - If Z is a class, all subobjects are zero initialized

User provided constructors

```
struct foo {
    foo() = default;  // foo() is not user provided
};

struct bar {
    bar();            // bar() is user provided
};
bar::bar() = default; // bar() is user provided

//ill-formed, const with no user-provided constructor
const int my_int;

//well-formed, has a user-provided constructor
const std::string my_string;

//ill-formed, const no user-provided constructor
const foo my_foo;

//well-formed, has a user-provided constructor
const bar my_bar;
```

A constructor quiz: answers

```
#include <iostream>

struct foo {
    foo() = default;  // foo() is not user provided
    int a;
};

struct bar {
    bar();            // bar() is user provided
    int b;
};

bar::bar() = default; // bar() is user provided

int main() {
    foo a{};
    bar b{};
    std::cout << a.a << ' ' << b.b;
}
```

A constructor quiz: answers

```
#include <iostream>

struct foo {
    foo() = default;  // foo() is not user provided
    int a;
};

struct bar {
    bar();            // bar() is user provided
    int b;
};

bar::bar() = default; // bar() is user provided

int main() {
    foo a{};    // zero initialized then value initialized
    bar b{};    // value initialized
    std::cout << a.a << ' ' << b.b;
}
```

A constructor quiz: answers

```
#include <iostream>

struct foo {
    foo() = default;  // foo() is not user provided
    int a;
};

struct bar {
    bar();            // bar() is user provided
    int b;
};

bar::bar() = default; // bar() is user provided

int main() {
    foo a{};    // zero initialized then value initialized
    bar b{};    // value initialized
    std::cout << a.a << ' ' << b.b;   // 0
}
```

A constructor quiz: answers

```
#include <iostream>

struct foo {
    foo() = default;  // foo() is not user provided
    int a;
};

struct bar {
    bar();            // bar() is user provided
    int b;
};

bar::bar() = default; // bar() is user provided

int main() {
    foo a{};    // zero initialized then value initialized
    bar b{};    // value initialized
    std::cout << a.a << ' ' << b.b;   // 0 undefined
}
```

Quote from the standard

N4140 (essentially C++14) [dcl.fct.def.default]/5:

Explicitly-defaulted functions and implicitly-declared functions are collectively called defaulted functions, and the implementation shall provide implicit definitions for them (12.1 12.4, 12.8), which might mean defining them as deleted. A function is user-provided if it is user-declared and not explicitly defaulted or deleted on its first declaration. A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed.

A constructor quiz: answers

```
#include <iostream>

struct foo {
    foo() = default;  // foo() is not user provided
    int a;
};

struct bar {
    bar();            // bar() is user provided
    int b;
};

bar::bar() = default; // bar() is user provided

int main() {
    foo a{};    // zero initialized then value initialized
    bar b{};    // value initialized
    std::cout << a.a << ' ' << b.b;   // 0 undefined
}
```

Some more initializations :)

```
int i;          // default
int i{};        // value
static int i;   // zero
static int i = constexpr_fun();  // constant

int i{1}, int j(42);   // direct
int i = 42, j = i;     // copy
int i = {42};          // copy-list
int i{42};             // direct-list

int iarr[3] = {0,1};   // aggregate iarr[2] == 0
int iarr[3] = {.1=2};  // aggregate
const int &iref = 42;  // reference

// static: zero- or const initialization
// dynamic: non static
// unordered: dynamic init. of class template static member
// ordered: dynamic init. Of other non-locals with static life
// implicit: default or value
```

https://i.imgur.com/3wlxtI0.mp4

More to read on initialization

[accu]
https://accu.org/index.php/journals/2379#FN01

[cppcore]
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-always

[cppref1] value-initialization
http://en.cppreference.com/w/cpp/language/value_initialization

[cppref2] default-initialization
http://en.cppreference.com/w/cpp/language/default_initialization

[cppref3] zero-initialization
http://en.cppreference.com/w/cpp/language/zero_initialization

_

A simple date class

- ```
  class date { ... };

  main()
  {
      date exam( 2019, 12, 15); // constructor with 3 int

      if ( exam < date(2020) ) { ... } // 2020.1.1
      else if ( exam < date() ){ ... } // today

      date exam2( “2019.12.15”); // constructor with string

  }
  ```

- How many constructors should we define?
  ```

A simple date class

```
class date
{
public:
    // default constructor
    date();
    // constructor
    date( int y, int m=1, int d=1);
    // explicite constructor
    explicit date( const char *s);
    // …
private:
    // …
    int year;
    int month;
    int day;
};
```

Where to define operators?

- a + b

  a.operator+(b)

  operator+(a,b)

- a = b, a[b], a(b1,b2,...), a-> only member

  a.operator= (b)

  a.operator[ ] (b)

  a.operator( ) (b1, b2, …)

  a.operator->()

Where to define operators?

- Theory says: bind with class
  - Member operators
- Some operators can't be members
  - `std::ostream& operator<<(std::ostream&,const X&)`
- Sometimes members creates unwanted dependences

  `std::getline(std::basic_istream&,std::basic_string&)`
- Sometime operators should be symmetric

Symmetry

```
class date
{
public:
    bool operator<(const date& rhs);
};

int main()
{
    date today;   // current date from OS


    if ( today < date(2016) )  // works

    if ( today < 2016 )        // works



};
```

Symmetry

```
class date
{
public:
    bool operator<(const date& rhs);
};

int main()
{
    date today;   // current date from OS


    if ( today < date(2016) )  // works

    if ( today < 2016 )        // works

    if ( 2016 < today )        // does not work
                               // if operator< is member
};
```

Symmetry

```
class date
{
public:
};
bool operator<(const date& lhs, const date& rhs);

int main()
{
    date today;   // current date from OS


    if ( today < date(2016) )  // works

    if ( today < 2016 )        // works

    if ( 2016 < today )        // works
                               // if operator< is gobal
};
```

Inline functions

```
// date.h
class date
{
public:
    // …
private:
    // …
    int year;
    int month;
    int day;
};

       bool operator<( date d1, date d2);
inline bool operator==(date d1,date d2)   { ... }
inline bool operator!=( date d1, date d2) { ... }
inline bool operator<=( date d1, date d2) { ... }
inline bool operator>=( date d1, date d2) { ... }
inline bool operator>( date d1, date d2)  { ... }
```

Creation of subobjects

```
// date.h
class date
{
public:
    date( int y, int m, int d);
private:
    // …
    int year;
    int month;
    int day;
};

date::date(int y, int m, int d) : day(d),month(m),year(y)
{ . . . }

// rearranged to declaration order:
date::date(int y, int m, int d) : year(y),month(m),day(d)
{ . . . }
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << ‘\n’; }


private:
    int a{1};
    int b{2};
    std::string s{“string”};
};

int main()
{
    Simple obj;          // ?
    return 0;
}
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << '\n'; }


private:
    int a{1};
    int b{2};
    std::string s{"string"};
};

int main()
{
    Simple obj;          // 12string
    return 0;
}
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << ‘\n’; }
    Simple(int aa, int bb) : a(aa), b(bb)
                { cout << a << b << s << ‘\n’; }
private:
    int a{1};
    int b{2};
    std::string s{“string”};
};

int main()
{
    Simple obj{5,6};   // ?
    return 0;
}
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << '\n'; }
    Simple(int aa, int bb) : a(aa), b(bb)
                { cout << a << b << s << '\n'; }
private:
    int a{1};
    int b{2};
    std::string s{"string"};
};

int main()
{
    Simple obj{5,6};   // 56string
    return 0;
}
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << ‘\n’; }
    Simple(int aa, int bb) : a(b ), b(bb)
                { cout << a << b << s << ‘\n’; }
private:
    int a{1};
    int b{2};
    std::string s{“string”};
};

int main()
{
    Simple obj{5,6};   // ?
    return 0;
}
```

Creation of subobjects C++11

```
class Simple
{
public:
    Simple( ) { cout << a << b << s << ‘\n’; }
    Simple(int aa, int bb) : a(b ), b(bb)
                { cout << a << b << s << ‘\n’; }
private:
    int a{1};
    int b{2};
    std::string s{“string”};
};

int main()
{
    Simple obj{5,6};   // 06string  a is undefined!
    return 0;
}
```

Destructors

- Should be virtual?
  - When used in polymorphic way
  - When there is multiple inheritance

```
class Base
{
public:
    virtual ~Base();
};
class Der : public Base
{
public:
    virtual ~Der();
};
```

```
void f()
{
    Base *bp = new Der;
    // …
    delete bp;
}
```

Arrays are not polymorphic!

```
struct Base
{
    Base() { cout << "Base" << " "; }
    virtual ~Base() { cout << "~Base" << endl; }
    int i;
};
struct Der : public Base
{
    Der() { cout << "Der" << endl; }
    virtual ~Der() { cout << "~Der" << " "; }
    int it[10]; // sizeof(Base) != sizeof(Der)
};
int main()
{
    Base *bp = new Der;
    Base *bq = new Der[5];

    delete    bp;
    delete [] bq;   // this causes runtime error
}
```

Cloning – “Virtual” constructors

- Constructors are not virtual
- But sometimes we need similar behavior

```
std::vector<Base*> source;
std::vector<Base*> target;

source.push_back(new Derived1());
source.push_back(new Derived2());
source.push_back(new Derived3());

// should create new instances of the
// corresponding Derived classes and
// place them to target
deep_copy( target, source);
```

Wrong approach

```
deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    target.push_back(new Base(**i)); // creates Base()
  }
}
```

Wrong approach 2

```
deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    if ( Derived1 *dp = dynamic_cast<Derived1*>(*i) )
      target.push_back(new Derived1(*dp));
    else if ( Derived2 *dp = dynamic_cast<Derived2*>(*i) )
      target.push_back(new Derived2(*dp));
    else if ( Derived3 *dp = dynamic_cast<Derived3*>(*i) )
      target.push_back(new Derived3(*dp));
  }
}
```

Cloning

```
class Base
{
public:
  virtual Base* clone() const = 0;
};

class Derived : public Base
{
public:
  virtual Derived* clone() const { return new Derived(*this); }
};

deep_copy( std::vector<Base*> &target, const std::vector<Base*> &source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    target.push_back((*i)->clone()); // inserts Derived()
  }
}
```

Assignment operator

```
class MyClass
{
public:
  MyClass& operator=(const MyClass& rhs);
  // ...
private:
  std::vector<int> v;
  // resources...
};

MyClass& MyClass::operator=(const MyClass& rhs)
{
  if ( &rhs != this )
  {
    // release old resources
    // allocate new resources
    // copy resources from rhs to new
    v = rhs.v;   // release+allocate+copy
  }
  return *this;
}
```

Assignment operator

```
class MyClass
{
public:
  MyClass& operator=(const MyClass& rhs);
  // ...
private:
  std::vector<int> v;
  // resources...
};

MyClass& MyClass::operator=(const MyClass& rhs)
{
  if ( &rhs != this )
  {
    MyClass temp(rhs);  // allocate+copy
    v.swap(temp.v);     // do not throw
  }                     // release old resource
  return *this;
}
```

Assignment operator

```
class MyClass
{
public:
  MyClass& operator=(const MyClass& rhs);
  // ...
private:
  std::vector<int> v;
  // resources...
};

MyClass& MyClass::operator=(MyClass rhs)  // allocate+copy
{
  if ( &rhs != this )
  {
    MyClass temp(rhs);
    v.swap(rhs.v);      // do not throw
  }                     // release old resource
  return *this;
}
```

Virtual assignment operator?

```
#include <iostream>

struct Base
{
  virtual Base&
  operator=(const Base& rhs)
  {
    std::cout<<"Base::operator="<<'\n';
    a = rhs.a;
    return *this;
  }
  int a;
};

struct Derived : public Base
{
  virtual Derived&
  operator=(const Base& rhs) override
  {
    std::cout<<"Derived::operator="<<'\
n';
    a = 42;
    b = 43;
    return *this;
  }
  int b;
};
```

```
int main()
{
  Derived d1;
  Derived d2;
  d1.a = 1; d1.b = 2;
  d2.a = 3; d2.b = 4;

  std::cout<<" d1 == "<<d1.a
                  << ","<<d1.b<<'\n';
  std::cout<<" d2 == "<<d2.a
                  << ","<<d2.b<<'\n';
  d1 = d2;
  std::cout<<" d1 == "<<d1.a
                  << ","<<d1.b<<'\n';
}
```

Virtual assignment operator?

```
#include <iostream>

struct Base
{
  virtual Base&
  operator=(const Base& rhs)
  {
    std::cout<<"Base::operator="<<'\n';
    a = rhs.a;
    return *this;
  }
  int a;
};

struct Derived : public Base
{
  virtual Derived&
  operator=(const Base& rhs) override
  {
    std::cout<<"Derived::operator="<<'\
n';
    a = 42;
    b = 43
    return *this;
  }
  int b;
};
```

```
int main()
{
  Derived d1;
  Derived d2;
  d1.a = 1; d1.b = 2;
  d2.a = 3; d2.b = 4;

  std::cout<<" d1 == "<<d1.a
                  << ","<<d1.b<<'\n';
  std::cout<<" d2 == "<<d2.a
                  << ","<<d2.b<<'\n';
  d1 = d2;
  std::cout<<" d1 == "<<d1.a
                  << ","<<d1.b<<'\n';
}
```

```
$ ./a.out
d1 == 1,2
d2 == 3,4
Base::operator=
d1 == 3,4
```

Delegated constructors C++11

```
// C++98
class X
{
   int a;
   validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
   X(int x) { validate(x); }
   X() { validate(42); }
   X(string s) { int x = lexical_cast<int>(s); validate(x); }
   // ...
};

// C++11
class X
{
   int a;
public:
   X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
   X() : X{42} { }
   X(string s) : X{lexical_cast<int>(s)} { }
   // ...
};
```

Use constructors C++11

```
class Base
{
public:
    void f(double);   // overloaded member function
    void f(int);      // overloaded member function

    Base(double);     // overloaded constructors
    Base(int);        // overloaded constructors
    // ...
};

class Derived : public Base
{
public:
    using Base::f;    // lift Base's f into Derived scope – works in C++98
    void f(char);     // provide a new f
    void f(int);      // prefer this f to Base::f(int)

    using Base::Base; // lift Base constructors Derived scope – C++11 only
    Derived(char);    // provide a new constructor
    Derived(int);     // prefer this constructor to Base::Base(int)
    // ...
};
```

Use constructors C++11

```
#include <iostream>

struct Base
{
  Base(int i)    { std::cout << "Base(int) " << i << std::endl; }
  Base(double d) { std::cout << "Base(double) " << d << std::endl; }
  void f(int i)  { std::cout << "Base::f(int) " << i << std::endl; }
  void f(double d) { std::cout << "Base::f(double) " << d << std::endl; }
};
struct Der : Base
{
  using Base::Base;
  using Base::f;
  Der(char c) : Base(c)   { std::cout << "Der(char) " << c << std::endl; }
  Der(double d) : Base(d) { std::cout << "Der(double) " << d << std::endl; }
  void f(char c) { std::cout << "Der::f(char) " << c << std::endl; }
};
int main()
{
  Der x1('x'), x2(1), x3(3.14);
  x1.f('x');
  x1.f(1);
  x1.f(3.14);
}
```

```
Base(int) 120
Der(char) x
Base(int) 1
Base(double) 3.14
Der(double) 3.14
Der::f(char) x
Base::f(int) 1
Base::f(double) 3.14
```

Use constructors: strong types

```
#include <iostream>

struct Date
{
  int year, month, day;   // no constructor: aggregate type
};

struct BornDate : Date { };  // still aggregate type

struct DeathDate : Date { }; // still aggregate type


void print( BornDate born, DeathDate death);


Int main()
{
  const BornDate  born  = { 1823, 1,  1}; // aggregate initialization
  const DeathDate death = { 1849, 7, 31}; // aggregate initialization

  print( born, death);
  // print( death, born);   // compile error

}
```

Uniform (braced) initialization C++11

```
double t[] = { 1, 2, 3.456, 99.99 };
vector<double> v = { 1, 2, 3.456, 99.99 };

int x1(0);    //
int x2 = 0;   // same as x2(0)
int x3{0};    // C++11, but do not narrowing
int x4 = {0}; // C++11, usually the same as x3{0}

std::atomic<int>  a1{0};  // ok
std::atomic<int>  a2(0);  // ok
std::atomic<int>  a3 = 0; // syntax error!  Non-copyable

class X
{
private:
  int x{0};  // ok, C++11
  int y = 0; // ok, C++11
  int z(0);  // syntax error!
};

C c2;      // Ok
C c1();    // Most vexing parse
C c3{};    // Ok
```

Initializer list

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
                                       {"Nygaard","Simula"},
                                       {"Richards","BCPL"},
                                      {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
  { {"Maurice","Vincent","Wilkes"},{1913, 1945, 1951, 1967, 2000} },
  { {"Martin","Ritchards"}, {1982, 2003, 2007} },
  { {"David","John","Wheeler"}, {1927, 1947, 1951, 2004} }
};

auto x1 = 5;    // deduced type is int
auto x2(5);     // deduced type is int
auto x3{ 5 };   // deduced type is int since C++17
auto x4{ 5, 6 };// error since C++17
auto x5 = { 5 };// std::initializer_list<int>

enum class E;
E e { 42 };     // ok, since C++17
```

Initializer list

```
vector<double> v = { 1, 2, 3.456, 99.99 }; // constructor
v = { 4., 6.};       // operator=
f({ 4., 6., 7.});    // void f( std::vector<double> )
for ( auto i : { 1, 2, 3, 4, 5} )  ...


const T[N];   // underlying temporary array
              // each element is copy initialized
              // except no narrowing conversion

class initializer_list<T>
{
public:
  // usual copy operations do shallow copy
private:
  T *begin;
  T *end;    // or size_t n;
};
```

Default and deleted constructors C++11

```
class X
{
private:
    X(const X&);              // pre-C++11
    X& operator=(const X&); // pre-C++11
};

class X : private boost::noncopyable
{
    // ...
};

class X
{
    X(const X&) = delete;              // C++11
    X& operator=(const X&) = delete; // C++11
};

class X
{
    X(const X&) = default;              // C++11
    X& operator=(const X&) = default; // C++11
};
```

Constructors and exceptions

```
class C
{
public:
    C(int i, int j)
    try
        : x(i), z(j)    // x(i) or z(j) may throw exception
    { }
    catch( ... )
    {
        // we get here if either x(i) or z(j) throws exception
        // if x(i) throws, then z uninitialized
        // if z(j) throws, then ~X::X() has already executed

        // what to do here ??
    }
private:
    X x;
    Z z;
};
```

Constructors and exceptions

```
class C
{
public:
    C(int i, int j)
    try
        : x(i), z(j)    // x(i) or z(j) may throw exception
    { }
    catch( ... )
    {
        // do something
        if ( ... )
            throw OtherException();
        // otherwise
        //    constructor will re-throw original exception
    }
private:
    X x;
    Z z;
};
```

Constructors and exceptions

```
class C
{
public:
    C(int i, int j)
    try
        : x(i), z(j)    // x(i) or z(j) may throw exception
    { }
    catch( ... )
    {
        // do something
        if ( ... )
            throw OtherException();
        // otherwise
        //    constructor will re-throw original exception
    }
private:
    X x;
    Z z;
};
```

Destructors must not throw!

Complex initialization

```
void f(int param, bool flag)
{
    const int ci1 = 10 * param + 3;

    const int ci2 = flag ? 2 * param : 2 + param;

    const int ci3 =  ?    /* very complex initialization */




};
```

Complex initialization

```
void f(int param, bool flag)
{
    const int ci1 = 10 * param + 3;

    const int ci2 = flag ? 2 * param : 2 + param;

    const int ci3 =  f(); /* very complex initialization */




};

int f(int param, bool flag)
{
    int value;
    /* very complex initialization of value */
    return value;
}
```

Complex initialization

```
void f(int param, bool flag)
{
    const int ci1 = 10 * param + 3;

    const int ci2 = flag ? 2 * param : 2 + param;

    const int ci3 =  [&]{ /* very complex initialization */

       int value;
       /* very complex initialization of value */
       return value;
    }();
};
```