سلسلة تعلم البرمجة بلغة ++C الحديثة

Learn Modern C++ Programming Course

إعداد المهندس أحمد الديب

#33: Class Basics 3

Class Object Initialization

```cpp
class Person {
 public:
  std::string name;
  int age;
  std::string city;
};


int main() {
  Person s1{"Tamer", 25, "Zagazig"};  // memberwise initialization
  Person s2{s1};                      // copy initialization
  Person none{};                      // default initialization
  Person no_init;
}
```

we can initialize objects of a class for which we have not defined a constructor using
- memberwise initialization,
- copy initialization, or
- default initialization.

For statically allocated objects, the rules are exactly as if you had used {}. However, for local variables and free-store objects, the default initialization is done only for members of class type, and members of built-in type are left uninitialized.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Initialization by Constructors

```
class Person {
 public:
  Person(std::string name, int age, std::string city)
      : name(name), age(age), city(city) {
    std::cout << "Constructor called" << std::endl;
  }
  std::string name;
  int age;
  std::string city;
};


int main() {
  Person s1{"Tamer", 25, "Zagazig"};  // Constructor initialization
  Person s2{s1};                      // copy initialization
  Person none{};                      // ERROR: default initialization
  Person no_init;                     // ERROR
}
```

Using the () notation, you can request to use a constructor in an initialization.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Default Constructors

```
class Person {
 public:
  Person(){};  // default constructor
  Person(std::string name, int age, std::string city)
      : name(name), age(age), city(city) {
    std::cout << "Constructor called" << std::endl;
  }
  std::string name;
  int age;
  std::string city;
};


int main() {
  Person s1{"Tamer", 25, "Zagazig"};  // Constructor initialization
  Person s2{s1};                      // copy initialization
  Person none{};                      // OK: default initialization
  Person no_init;                     // OK
}
```

A constructor that can be invoked without an argument is called a default constructor. A default argument can make a constructor that takes arguments into a default constructor.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Initializer-List Constructors

```
template <typename T>
class Array {
 public:
  Array(std::initializer_list<T> list) : _size(list.size()) {
    for (auto i = 0; i < list.size(); ++i) {
      _elem[i] = list.begin()[i];
    }
  }


 private:
  T* _elem;
  std::size_t _size;
};


int main() {
  Array arr1{1, 2, 3, 4, 5};
  Array arr2{10, 20, 30};
}
```

A constructor that takes a single argument of type std::initializer_list is called an initializer-list constructor.

The initializer list can be of arbitrary length but must be homogeneous. That is, all elements must be of the template argument type, T, or implicitly convertible to T.

Unfortunately, initializer_list doesn't provide subscripting.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Delegating Constructors

```
class Date {
 public:
  Date(int d, int m, int y) { validate(d, m, y); }
  Date(int d, int m) { validate(d, m, 2023); }
  void print();


 private:
  int _day, _month, _year;
  void validate(int d, int m, int y) {
    if ((d > 31) || (d < 1) || (m > 12) || (m < 1) || (y > 2023) || (y < 1900))
      throw std::invalid_argument{"Date not valid"};


    _day = d;
    _month = m;
    _year = y;
  }
};
```

If you want two constructors to do the same action, you can repeat yourself or define a function to perform the common action. Both solutions are common (because older versions of C++ didn't offer anything better).

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Delegating Constructors

```
class Date {
 public:
  Date(int d, int m, int y) {
    if ((d > 31) || (d < 1) || (m > 12) || (d < 1) || (y > 2023) || (y < 1900))
      throw std::invalid_argument{"Date not valid"};


    _day = d;
    _month = m;
    _year = y;
  }
  Date(int d, int m) : Date{d, m, 2023} {}
  void print();


 private:
  int _day, _month, _year;
};
```

That is, a member-style initializer using the class's own name (its constructor name) calls another constructor as part of the construction.

You cannot both delegate and explicitly initialize a member.

Delegating by calling another constructor in a constructor's member and base initializer list is very different from explicitly calling a constructor in the body of a constructor.

```
Date(int d, int m) : {Date{d, m, 2023}}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Slicing

```cpp
struct Base {
  int b;
  Base() {}
  Base(const Base&) {
    std::cout << "Call base class copy constructor" << std::endl;
  }
  // ...
};


struct Derived : Base {
  int d;
  Derived() {}
  Derived(const Derived&) {
    std::cout << "Call member derived copy destructor" << std::endl;
  }
  // ...
};


void naive(Base* p) {
  Base b2 = *p;  // slice
  // ...
}


int main() {
  Derived d;
  naive(&d);
  Base bb = d;  // slice
  // ...
}
```

slicing object from type 'Derived' to 'Base' discards 4 bytes of state clang-tidy(cppcoreguidelines-slicing)

View Problem (⌥F8)   No quick fixes available

```
A pointer to a derived class implicitly
converts to a pointer to its public base class.
When applied to a copy operation, this simple
and necessary rule leads to a trap for the
unwary.

- Prohibit copying of the base class: delete
the copy operations.
- Prevent conversion of a pointer to a derived
to a pointer to a base: make the base class a
private or protected base.

The former would make the initializations of b2
and bb errors; the latter would make the call
of naive() and the initialization of bb errors.
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Explicit Defaults

```
class Person {
 public:
  Person(std::string name, std::string city, std::string company,
         std::string position)
      : _name(name), _city(city), _company(company), _position(position) {}
  Person() = default;
  ~Person() = default;
  Person(const Person&) = default;
  Person(Person&&) = default;
  Person& operator=(const Person&) = default;
  Person& operator=(Person&&) = default;

 private:
  std::string _name, _city, _company, _position;
};
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

deleted Functions

```
struct Base {
  int b;
  Base() {}
  Base& operator=(const Base&) = delete;  // disallow copying
  Base(const Base&) = delete;
  Base& operator=(Base&&) = delete;   // disallow moving
  Base(Base&&) = delete;
  // ...
};

template <class T>
T* clone(T* p)  // return copy of *p
{
  return new T{*p};
};

// don't try to clone a Foo
Foo* clone(Foo*) = delete;
```

we can delete any function that we can declare. For example, we can eliminate a specialization from the set of possible specializations of a function template. Another application is to eliminate an undesired conversion.

```
struct Z {
  // ...
  Z(double);         // can initialize with a double
  Z(int) = delete;   // but not with an integer
};
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

deleted Functions

```
class Not_on_stack {
  // ...
  ~Not_on_stack() = delete;
};


class Not_on_free_store {
  // ...
  void* operator new(size_t) = delete;
};
```

A further use is to control where a class can be allocated.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Thank you