Язык C++

Template specialization. Smart Ptr

Специализация шаблона класса

```
template<class T>
struct Boo {
   void foo() {
        std::cout << "foo" << std::endl;
   }
};

template<>
struct Boo<int> {
   void foo() {
      std::cout << "foo(int)" << std::endl;
   }
};
```

Full template specialization

1. Шаблон функции
2. Шаблон класса
3. Шаблон переменной
4. Шаблона функции класс
5. Шаблона члена класса
6. ….

Специализация шаблона класса

```
template<class T>
struct Boo {
   void foo() {
        std::cout << "foo" << std::endl;
   }
   void func () {};
};

template<>
struct Boo<int> {
   void foo() {
      std::cout << "foo(int)" << std::endl;
   }
};
```

Специализация шаблона класса

```
int main() {
    std::vector<bool> bv;
    std::vector<int> bi;
    return 0;
}
```

Специализация шаблона класса

```
template<class T>
struct is_float {
  static bool value() {  return false;  }
};

template<>
struct is_float<float> {
  static bool value() { return true; }
};

template<class T>
static bool is_float_v = is_float<T>::value();
```

Специализация шаблонов функций

```
template<class T>
void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}

struct SomeStruct {};
```

Специализация шаблонов функций

```
template<>
void swap<SomeStruct>(SomeStruct& a, SomeStruct& b) {
  std::cout << "swap for SomeStruct with template" << std::endl;
}

void swap(SomeStruct& a, SomeStruct& b) {
  std::cout << "swap for SomeStruct without template" << std::endl;
}
```

Специализация шаблонов функций

```
template<class T>
void swap(std::vector<T>& x, std::vector<T>& y) {
  std::cout << "vector swap" << std::endl;
  x.swap(y);
};
```

Специализация шаблонного члена класса

```
struct SomeStruct {
  template<class T>
  void func(const T& x) {
      std::cout << x << std::endl;
  }

  void func(int x) {
      std::cout << "int" << std::endl;
  }
};
```

Частичная специализация

```
template<class T, class U>
struct Boo {
  void foo() { std::cout << "A" << std::endl; }
  void func () {};
};

template<class U>
struct Boo<int, U> {
  void foo() { std::cout << "B" << std::endl; }
};

template<>
struct Boo<int, int> {
  void foo() { std::cout << "C" << std::endl; }
};
```

RAII

- **Resource acquisition is initialization**
- Захват ресурса - есть инициализация
- Обеспечивает инкапсуляция ресурса и инвариант состояния
- Безопасна к исключениям для объектов лежащих на стеке
- Применяется для указателей, мьютексов, файлов,....

auto_ptr

```
struct Boo {
   Boo() { std::cout << "Boo()\n";}
   ~Boo() {std::cout << "~Boo()\n";}
};

void func() {
   Boo* b = new Boo();
   throw std::runtime_error("Error");
   delete b;
}
```

auto_ptr

```
template<class T>
class auto_ptr {
public:
   auto_ptr(T* ptr = nullptr)
      :ptr_(ptr)
   {}

   ~auto_ptr() {
       delete ptr_;
  }
private:
  T* ptr_;
};
```

auto_ptr

```
void func() {
    auto_ptr<Boo> b {new Boo()};
    throw std::runtime_error("Error");
}

int main() {
    try {
        func();
    }
    catch(std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

auto_ptr

```
void func() {
    auto_ptr<Boo> b {new Boo()};
    auto_ptr<Boo> p = b;
    throw std::runtime_error("Error");
}
```

```
auto_ptr(auto_ptr& other)
    : ptr_(other.release()){
}

T* release() {
    T* tmp = ptr_;
    ptr_ = nullptr;
    return tmp;
}
```

auto_ptr

```
void func() {
    auto_ptr<Boo> a {new Boo()};
    auto_ptr<Boo> b {new Boo()};
    a = b;
    throw std::runtime_error("Error");
}
```

```
auto_ptr& operator=(auto_ptr& other) {
      if(ptr_ != other.ptr_) {
          delete ptr_;
          ptr_ = other.release();
      }
   }
```

auto_ptr

```
void func() {
    auto_ptr<Boo> a {new Boo()};
    auto_ptr<Boo> b {new Boo()};
    a = b;
    throw std::runtime_error("Error");
}
```

```
auto_ptr& operator=(auto_ptr& other) {
      if(ptr_ != other.ptr_) {
          delete ptr_;
          ptr_ = other.release();
      }
   }
```

auto_ptr

```
void func() {
    auto_ptr<Boo> a {new Boo()};

    a->func();
    (*a).func();

    throw std::runtime_error("Error");
}
```

```
T* operator->() const {
    return ptr_;
}

T& operator*() const {
    return ptr_;
}
```

auto_ptr

1. В <memory> есть std::auto_ptr
2. **deprecated in C++11**
3. **removed in C++17**

auto_ptr

```
int main() {
    auto_ptr<Boo> b {new Boo()};
    std::vector<auto_ptr<Boo>> boos(1);

    boos[0] = b;
    boos[0]->func();
    auto_ptr<Boo> a = boos[0];
    a->func();

    //b->func();         // Segmentation fault
    //boos[0]->func();   // Segmentation fault

    return 0;
}
```

Smart Pointer

1. unique_ptr
2. shared_ptr
3. weak_ptr

unique_ptr

1. Во многом похож на auto_ptr
2. Нет конструктора копирования
3. Нет оператора присваивания
4. make_unique
5. deleter
6. std::default_deleter

unique_ptr

```
struct FileDeleter{
   void operator()(FILE* file){

       if(file != nullptr) {
           fclose(file);
           file = nullptr;
       }
   }
};

int main() {
   std::unique_ptr<FILE, FileDeleter> f {fopen("temp.txt", "w")};
   return 0;
}
```

std::shared_ptr

1. Атомарный счетчик
2. Копирование увеличивает счетчик
3. Деструктор уменьшает
4. Уничтожение при счетчике = 0
5. std::make_shared

std::shared_ptr

```
void func(){
  std::shared_ptr<Boo> p1 = std::make_shared<Boo>();
  std::shared_ptr<Boo> p2 = p1;

  p1->func();
  p2->func();

  std::cout << p1.use_count() << std::endl;
}
```

std::shared_ptr

```
struct A;

struct B {
  B() { std::cout << "B\n";}
  ~B() { std::cout << "~B\n";}
  std::shared_ptr<A> ptr;
};

struct  A {
  A() { std::cout << "A\n";}
  ~A() { std::cout << "~A\n";}
  std::shared_ptr<B> ptr;
};
```

```
void func(){
  std::shared_ptr<A> a {new A()};
  std::shared_ptr<B> b {new B()};
  a->ptr = b;
  b->ptr = a;

  // nothing will be deleted
}
```

std::weak_ptr

1. Не владеет объектом
2. Может вернуть shared_ptr через Lock
3. Знает количество
4. **user_count**
5. **expired**
6. **bad_weak_ptr**

std::enable_shared_from_this

1. CRTP
2. Позволяет создать shared_ptr внутри методов объектов в shared_ptr
3. Кидает bad_weak_ptr если объект не shared_ptr