Язык C++

Cast, CRTP

Cast

- Implicit
- Explicit
  - const_cast
  - static_cast
  - dynamic_cast
  - reinterpret_cast
  - C-style cast

Implicit cast

```
int main(int, char**) {
    double d = -12.3456789;
    std::cout << d << std::endl;
    float f = d;
    std::cout << f << std::endl;
    int i = d;
    std::cout << i << std::endl;
    uint32_t ui = d;
    std::cout << ui << std::endl;
    char ch = d;
    std::cout << ch << std::endl;
    return 0;
}
```

Implicit cast

```
int main(int, char**) {
    std::vector<bool> v(2147483647, 0);
    size_t cnt = 0;
    for(int i = 0; i < v.size(); ++i) {
        if(v[i] == 0)
            ++cnt;
        if(cnt % 1000000000 == 0)
            std::cout << cnt << std::endl;
    }

    std::cout << cnt << std::endl;
}
```

C-style cast

```
int main(int, char**) {
    int i = 1;
    std::cout << *(double*)&i << std::endl;

    return 0;
}
```

C-style cast

```
struct Foo {
   int i  = 64;
};

struct Boo {
   std::string str = "abc";
};

void func(Boo* b) {
   std::cout << b->str << std::endl;
}
```

```
int main(int, char**) {
   Boo b;
   Foo f;

   func(&b);
   func((Boo*)&f);
   return 0;
}
```

const_cast

- Убирает const или volatility с переменное
- Может преобразовывать указатели на одинаковые типы данных
- Может преобразовывать ссылки

const_cast

```
int main () {
   const int i = 0;
   const int* cpi = &i;

   int* pi = const_cast<int*>(cpi);
   *pi = 100500;

   std::cout << i << std::endl;

   const char* str = "Hello world!";
   char* s = const_cast<char*>(str);
   // s[0] = 'A'; // Undefined behaviour !!
}
```

const_cast

```
class Foo {
public:
  int Value() const {
      const_cast<Foo*>(this)->counter_++;
      return value_;
  }
private:
  int value_ = 0;
  int counter_ = 0;
};
```

static_cast

- Пытается выполнить преобразование с помощью конструкторов и операторов приведения
- Работает в момент compile-time
- Работает для стандартных типов
- Работает для приведения указателей из одной иерархии
- Может приводить из указателя на void

static_cast

```
struct Foo {
   Foo(float f) {};
};

struct Boo {
   Boo(const std::string&) {}

   operator int() {
       return 2;
   }
};

int main(int, char**) {
   Boo b("12345");
   Foo f = static_cast<Foo>(b);
   return 0;
}
```

static_cast

```
struct Base {
  void func() {
      std::cout << "Base\n";
  }
};


struct Derrived : public Base {
  void func() {
      std::cout << "Derrived\n";
  }
};
```

static_cast

```
int main() {
  Derrived d;
  Base& b = d;
  b.func();

  Derrived& d1 = static_cast<Derrived&>(b);
  d1.func();

  void* t = new Derrived();
  static_cast<Derrived*>(t)->func();
}
```

static_cast

```
struct Base {
   int i = 1;
};

struct Derrived : public Base {
   int j = 2;
   void func() {
       std::cout << j << std::endl;
   }
};

int main(int, char**) {
   Base b;
   Derrived& d = static_cast<Derrived&>(b);

   d.func();
   return 0;
}
```

dynamic_cast

- Преобразует указатели и ссылки вниз и вверх по иерархии
- RTTI
- std::bad_cast

dynamic_cast

```
struct Base {
    int i = 1;
    virtual ~Base() {};
};

struct Derrived : public Base {
    int j = 2;
    void func() {
        std::cout << j << std::endl;
    }
};
```

```
int main(int, char**) {
    Base* b = new Base();
    Derrived* d = dynamic_cast<Derrived*>(b);

    std::cout << d << std::endl;
    d->func();
    return 0;
}
```

```
int main(int, char**) {
    Base b;
    Derrived& d = dynamic_cast<Derrived&>(b);

    d.func();
    return 0;
}
```

reinterpret_cast

- Позволяет кастовать несовместные типы
- Использует исключительно побитовое представление

reinterpret_cast

```
int main() {
  std::ofstream out("temp.txt");
  SPoint* p = new SPoint{1, 2};
  char* ch = reinterpret_cast<char*>(p);
  out.write(reinterpret_cast<const char*>(p), sizeof(SPoint));
  out.close();

  std::ifstream in("temp.txt");
  char buffer[sizeof(SPoint)];
  in.read(buffer, sizeof(SPoint));
  in.close();
  SPoint* p1 = reinterpret_cast<SPoint*>(buffer);
}
```

Curiously Recurring Template Pattern

```
template<typename T>
class Base {
public:
};

class Derrived : public Base<Derrived> {
};
```

Curiously Recurring Template Pattern

```
template<typename T>
struct Counter {
  static int created;
  Counter() {
      ++created;
  }
  Counter(const Counter&) {
      ++created;
  }
};

template<typename T> int Counter<T>::created = 0;
```

Curiously Recurring Template Pattern

```
template<typename T>
class Base {
public:
  void doSomething() {
      T* derrived = static_cast<T*>(this);
  }
};

class Derrived : public Base<Derrived> {
};
```

Curiously Recurring Template Pattern

```
template<typename T>
class Cloneable {
public:
  T clone() const {
      return T{static_cast<const T&>(*this)};
  }
};
```

Curiously Recurring Template Pattern

```
template<class ConcreateAnimal>
class Animal {
public:
  std::string who() const {
      return static_cast<const ConcreateAnimal*>(this)->who();
  }
};

template<class T>
void who_am_i(Animal<T>& animal) {
  std::cout << animal.who() << std::endl;
}
```

Curiously Recurring Template Pattern

```
class Dog : public Animal<Dog> {
public:
  std::string who() const {
      return "dog";
  }
};
class Cat : public Animal<Cat> {
public:
  std::string who() const {
      return "cat";
  }
};
```

Other cast

itoa

to_string

bit_cast

to_chars