Лекция 4

Константные выражения

constexpr

- constexpr сообщает, что выражение должно быть рассчитано на этапе компиляции
- Может быть применено к переменным и функциям

```
constexpr int sum(int a, int b){
    return a + b;
}

int main(){
    constexpr int val1 = 14;
    constexpr int val2 = sum(3,4);
    int n = 10;
    constexpr int val3 = sum(n, 4); //Ошибка
    constexpr int val4 = sum(val1, val2);
    int val5 = sum(n, val1);
}
```

constexpr класс

```
class ConstExprClass{
    int value;
public:
    constexpr ConstExprClass(int value):value(value){}
    constexpr int getValue() const{
        return value > 0 ? value : 0;}
    void increase(){
        value++;}
};

int main(){
    constexpr ConstExprClass obj{6};
    std::cout << obj.getValue() << '\n';
    ConstExprClass obj2{7};
    obj2.increase();
    std::cout << obj2.getValue() << '\n';
    return 0;
}
```

constexpr ограничения переменных

- Могут быть переменные, для которых соблюдаются условия :
  1. Скалярные типы
  2. Указатели
  3. Массив скалярных типов
  4. Класс, в котором:
     - Деструктор по умолчанию
     - Все нестатические поля – литеральные типы данных
     - Хотя бы один constexpr конструктор или их отсутствие

constexpr ограничения функций

- Могут быть функции, для которых соблюдаются условия:
  1. Должны быть не virtual
  2. Должны возвращать литеральный тип
  3. Все аргументы должны иметь литеральный тип
  4. Тело функции может содержать только:
     - static_assert
     - typedef и using
     - Ровно один return, который содержит constexpr выражение

Перегрузка операторов

Оператор

- Оператор – функция, обозначенная специальным символом
- Сигнатура оператора такая же, как у функций, но с ключевым словом **operator #**, где # знак оператора
- Существуют унарные и бинарные операторы

Таблица операторов

Оператор	Тип по смыслу	По кол-ву аргументов
+ - * / %	Арифметические	Бинарные
+= -= *= /= %=		Унарные
+a -a	Смена знака	Унарные
++a --a	Префиксный инкремент	Унарные
a++ a--	Постфиксный инкремент	Унарные
&& || !	**Логические**	**Бинарные**
& | ^ << >>	Битовые	Бинарные
~		Унарные
=	Присваивание	Бинарный
== != < > <= >=	Сравнение	Бинарные
<< >>	Вывод в поток	Бинарные

Специальные операторы

- ***a*->** – доступ к полям по указателям – перегружать не рекомендуется
- ***a*.** – доступ к полям – перегружать нельзя
- ? : – тернарный оператор – перегружать нельзя
- :: – доступ к полю – перегружать нельзя
- () – вызов функции
- (type) – приведение к типу
- new – выделение памяти – перегружать не рекомендуется
- delete – освобождение памяти – перегружать не рекомендуется

Оператор присваивания

- Возвращает ссылку на объект класса
- Внутри оператора возвращается *this

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    MyInt& operator = (const MyInt& val){
        if(this != &val)
            this->i = val.i;
        return *this;
    }
};
```

Унарные арифметические операторы

- Принимают один аргумент
- Видоизменяют существующий объект
- Принято возвращать ссылку на текущий объект

```
MyInt& operator +=(const MyInt& val){
        this->i += val.i;
        return *this;
    }
```

Бинарные арифметические операторы

- Объявляются с модификатором friend
- Принимают 2 аргумента – левый и первый операнд
- Создают новый объект
- Возвращают результат по значению

```
friend MyInt operator + (const MyInt& val1, const MyInt& val2){
    return MyInt(val1.i + val2.i);
}
friend MyInt operator + (const MyInt& val1, const int& val2){
    return MyInt(val1.i + val2);
}
```

Унарные операторы

- Не принимают аргумента
- Возвращают ссылку на текущий объект

```
MyInt& operator -(){
    this->i = - this->i;
    return *this;
}
MyInt& operator +(){
    return *this;
}
```

Операторы инкремента и декремента

- Префиксный оператор не принимает аргументов и сначала меняет объект, потом возвращает ссылку на него
- Постфиксный инкремент принимает фиктивный аргумент, создает копию объекта, меняет текущий объект и возвращает копию по значению

```
MyInt& operator ++(){
        this->i += 1;
        return *this;
    }
MyInt operator ++(int){
    MyInt temp(this->i);
    this->i += 1;
    return temp;
}
```

Логические операторы

```
friend bool operator == (const MyInt& val1, const MyInt& val2){
    return (val1.i == val2.i); //Необходимо реализовать
}
friend bool operator != (const MyInt& val1, const MyInt& val2){
    return !(val1 == val2);
}
friend bool operator < (const MyInt& val1, const MyInt& val2){
    return (val1.i < val2.i); //Необходимо реализовать
}
friend bool operator > (const MyInt& val1, const MyInt& val2){
    return (val2.i < val1.i);
}
friend bool operator <= (const MyInt& val1, const MyInt& val2){
    return !(val1.i > val2.i);
}
friend bool operator >= (const MyInt& val1, const MyInt& val2){
    return !(val1.i < val2.i);
}
```

Оператор приведения к типу

- Особенность – не имеет возвращаемого типа
- Можно приводить к любому типу, известному в месте объявления оператора

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    operator int() const{
        return this->i;
    }
};
```

Перегрузка оператора ввода

- Левым операндом является ссылка на istream
- Возвращает по ссылке поток, из которого происходило чтение
- Правый операнд должен быть не константным

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    friend std::istream& operator>>(std::istream& in, MyInt& obj){
        in >> obj.i;
        return in;
    }
};
```

Перегрузка вывода в поток

- Левым операндом является ссылка на ostream
- Возвращает по ссылке поток, в который происходила запись

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    friend std::ostream& operator<<(std::ostream& out, const MyInt& obj){
        out << obj.i;
        return out;
    }

};
```

Операторы new и delete

- Увеличение производительности за счёт кеширования
- Выделение памяти сразу под несколько объектов
- Реализация собственного сборщика мусора
- Вывод логов выделения и освобождения памяти
- Реализация своего placement new
- Выделение памяти без исключений
- Собственные формы new

Рекомендации по перегрузке

Оператор	**Рекомендуемая форма**
Все унарные операторы	Член класса
+= -= /= *= ^= &= и т.д	Член класса
= () [] -> ->*	Обязательно член класса
Остальные бинарные операторы	Не член класса

Пользовательские литералы

Пользовательские литералы

- Пользовательские литералы определяются как
  operator "" suffix_identifier
- Позволяют переводить какой-либо литерал к пользовательскому типу

```
constexpr long double operator "" _deg (long double deg){
    return deg * 3.14159265358979323846264L / 180;
}
```

Ограничение на пользовательские литералы

- Оператор можно применять к типам:
  - unsigned long long int
  - long double
  - char, wchar_t, char8_t (c++20), char16_t, char32_t
  - const char *
  - (const char * , std::size_t)

Функторы

Оператор вызова функции

- Класс, для которого определен оператор вызова функции, называется функтором

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    MyInt& operator ()(int val){
        this->i *= val;
        return *this;
    }
    void print(){
        std::cout << i << '\n';
    }
};
```

Пример функтора 1

```
class SQR{
    double a;
    double b;
    double c;
public:
    SQR(double a = 1, double b = 0, double c = 0):a(a),b(b),c(c){}
    double operator()(double x = 0){
        return a*x*x + b*x + c;
    }
};

int main(){
    SQR f(1,-2,1);
    std::cout << f(5) << " " << f(1) << '\n';
    return 0;
}
```

Пример функтора 2

```
class F{
    int val;
public:
    F(int val):val(val){}
    F& operator()(int x){
        this->val += x;
        return *this;
    }
    operator int(){
        return this->val;
    }
};

int main(){
    std::cout << F(6)(4)(3)(-1);
}
```

Пример функтора 3

```
class Yield{
    std::vector<double> m_result;
public:
    double operator()(double val){
        double e_val = exp(val);
        m_result.push_back(e_val);
        return e_val;
    }
    std::vector<double> result() const{
        return m_result;
    }
    double operator[](size_t i){
        return m_result.at(i);
    }
};
```

Функторы вместо lambda

- Функторы также могут использоваться вместо lambda-выражений

```
Functor func_obj;
std::vector<int> v = {-1,1,2};
std::for_each(v.begin(), v.end(), [](const int& val){std::cout << val << ' ';});
std::for_each(v.begin(), v.end(), func_obj);
```

std::mem_fn

- mem_fn позволяет преобразовать метод класс в функцию вне класса

```
struct Foo {
    void display_greeting() {
        std::cout << "Hello, world.\n";
    }
    void display_number(int i) {
        std::cout << "number: " << i << '\n';
    }
    int data = 7;
};
int main() {
    Foo f;
    auto greet = std::mem_fn(&Foo::display_greeting);
    greet(f);
    auto print_num = std::mem_fn(&Foo::display_number);
    print_num(f, 42);
    auto access_data = std::mem_fn(&Foo::data);
    std::cout << "data: " << access_data(f) << '\n';
}
```

std::function

- Полиморфная оболочка для функций

```
void proceedOperation(double val1, double val2,
       std::function<double(double,double)> f){
    std::cout << "Result is " << f(val1, val2) << '\n';
}
```

std::function

```
class Functor{
    double value;
public:
    Functor(double val):value(val){}
    double operator()(double val1, double val2){
        return val1 + val2 + value;
    }};

double sum(double val1, double val2){
    return val1 + val2;}

void proceedOperation(double val1, double val2, std::function<double(double,double)> f){
    std::cout << "Result is " << f(val1, val2) << '\n';}

int main(){
    double a = 0.4, b = -1.6;
    Functor f{3.4};
    proceedOperation(a, b, f);
    proceedOperation(a, b, sum);
    proceedOperation(a, b, [](double a, double b){return a - b;});
    std::function<double(double,double)> saved_func = sum;
}
```

Лямбда-функции

Лямбда-функции

- Является анонимной функцией
- Можно хранить в переменных
- Обычно используются для однострочных и редко используемых функций

```
int main(){
    [](){}; //Пустая лямбда
    [](){std::cout << "Hello, World!\n";}(); //Объявляем и сразу вызываем
    //Соханяем в переменную
    auto f = [](){std::cout << "Goodbye, World!\n";};
    f(); //Вызываем из переменной
    return 0;
}
```

Передача аргументов

- Аргументы можно передавать как и в обычной функции

```
int main(){
    int a = 10, b = 7;
    auto f = [](int a, int b){return a + b;};
    std::cout << f(a,b) << '\n';
    std::cout << ([](int a, int b){return a - b;}(a,b)) << '\n';
    return 0;
}
```

Захват переменных

- Можно производить захват переменных из области видимости, где объявляется лямбда

```
int main(){
    int a = 10, b = 7;
    auto f = [=]{return a + b;}; //Захват по значению (read-only)
    std::cout << f() << '\n';
    std::cout << ([&]{return a - ++b;}()) << '\n'; //Захват по ссылке
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Захват переменных

- Можно указывать, какие переменные и как захватывать

```
int main(){
    int a = 10, b = 7;
    auto f = [a](int b){return a + b;}; //Захват по значению (read-only)
    std::cout << f(3) << '\n';
    //Захват по ссылке
    std::cout << ([&b](int a){return a - ++b;}(22)) << '\n';
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Захват переменных в классе

- Лямбды не знают про поля класса
- Можно захватить this и дать доступ к полям класса

```
class A{
    int x;
public:
    void func(){
        double z;
        [this]{x *= 2; /*z *= 2;*/}();
    }
};
```

Возвращаемый тип

- По умолчанию тип выводится из результата операции
- Можно явно задать тип после списка аргументов

```
int main(){
    double a = 10.5, b = 7.2;
    auto f = [=]()->int {return a + b;};
    std::cout << f() << '\n';
    std::cout << ([=]()->int {return a - b;}()) << '\n';
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Пример использования (1)

```
class SumResult{
    int value;
public:
    SumResult():value(0){}
    auto getFunc(){
        return [this](int add_value){value += add_value;};
    }
    int getResult(){
        return value;
    }
};

int main(){
    SumResult res;
    std::vector<int> vec = {1,2,-3,5,6};
    std::for_each(vec.begin(), vec.end(),res.getFunc());
    std::cout << res.getResult();
    return 0;
}
```

Пример использования (2)

```
class SumResult{
    int value;
public:
    SumResult():value(0){}
    void apply(std::function<void(int&)> f){
        f(value);}
    int getResult(){
        return value;
    }};

int main(){
    SumResult res;
    int a = 10;
    int b = 4;
    res.apply([](int& value){value += 2;});
    res.apply([=](int& value){value *= a;});
    res.apply([&](int& value){value -= b++;});
    std::cout << b << ' ' << res.getResult();
}
```

Сложный пример (1)

```
#include<iostream>
#include<functional>

typedef std::function<void(int& )> log_func;

class Logger{
public:
    virtual log_func getLogFunc() = 0;
};

class ConsoleLogger: public Logger{
public:
    log_func getLogFunc(){
        return [](int& value){std::cout << value << '\n';};
    }
};
```

Сложный пример (2)

```
class Operator{
    int value;
    std::vector<log_func> funcs_to_do;
    void proceedFuncs(){
        for(auto f: funcs_to_do){
            f(value);
        }
    }
public:
    Operator(int value = 0):value(value){}
    void addFunc(log_func f){
        funcs_to_do.push_back(f);
    }
    void add(int value){
        this->value += value;
        this->proceedFuncs();
    }
    void mul(int value){
        this->value *= value;
        this->proceedFuncs();
    }
};
```

Сложный пример (3)

```
int main(){
    Operator op(1);
    Logger* logger = new ConsoleLogger();
    op.addFunc(logger->getLogFunc());
    op.addFunc([](int& value){value /= 2;});
    op.add(3);
    op.add(5);
    op.mul(4);
    delete logger;
    return 0;
}
```