Седьмой семинар по программированию

cmake — (compiler make) программа для компиляции make файла. Требует проверки версии и позволяет множество надстроек. Позволяет сгенерировать собиратель файла для какой-либо платформы. Кросс - платформенная штука. Но с ним не так удобно работать, так как из-за высокоуровнетости и кроссплатформинга он менее гибкий, чем make. Cmake не работает в той же папке, где он находится. Для этого нужно создавать отдельную папку.

Приведение типов

Явное — неявное приведение типов языка.

Явное:
(type)(expression) — язык C и C++
type(expression) — в C++

Неявное:
char a = sqrt(5);
Это может вызывать ошибку, если не уследить.

Так что добавили auto:
auto a = sqrt(5);

Осторожней работаем с типами данных и пытаемся уследить. Были даже случаи того, что ошибки программистов приводили к смертям: рентгеновские аппараты давали смертельную дозу, ПВО промахивалось.

sizeof(char) = 1; //всегда вызывает 1, в C++ более строгое отношение к типам данных.

Почему это так? Чтобы не потерять обратную совместимость.

```
class fruit {
public:
        explicit fruit (std::string&) = «default»; //explicit запретит неявное приведение //1
        operator=(fruit&); //перегрузка операторов без проблелов //2
        explicit operator  std::string( ) const; // аналогично //3
        int ind;
};
fruit apple(«apple»); //const char* или const char[], sizeof(…) от них будет разным
                        // char переводится в string, а это уже подставляется в констр.
fruit mango(« ») = «do mango»; //вновь

//явно:
fruit apple = fruit(«mango»); //йова
std::string str = fruit(«mango»); // не рабоает с explicit, тк может быть неявно
std::string str = std::string(fruit(«mango»)); // работает с explicit, контроль приведения

В c++ добавили приведение к другим типам данных:
operator  Typename( ) const; //здесь пробел. Объект класса не поменяется, тк const
```

Пропись привода типа данных:

```
operator std::string( ) const {
        return name;
}
```

Мы с вами уже познакомились с открытым наследованием. Повторили пару слов про protected, private, public. Познакомимся теперь с private наследованием.

:public

наследование интерфейса

отношение является

fruit(std::string& str):
(1): name (str)
(2): std::string(str)
{
}

String getName()
{
(1)return name;
*(2)return string(*this)*
}

:private class Name

скрыт интерфейс полностью

отношение включает

Если не указать, как наследоваться, то наследование будет приватным

(1)class fruit {
std::string name;
};
==
(2)class fruit: private string {
//public:?
fruit(string&);
string getName( );
};

Доступ к методам и объектам базового класса

ostream operator<<(ostream& os, const string);

ostream operator<<(ostream& os, const fruit fr) {
*os << string(fr) << '\n'; /*в метод передается указатель на сам string*/*
return os;
}

Если у нас есть методы protected, то при private наследовании можно использовать их.

При приватном наследовании есть интересные оптимизации, суть в том, что объект может быть пустым. Пустые объекты в C++ занимают какое-то пространство. Что ссылка на стринг, что ссылка на фрут у нас будет… Таким образом, есть некоторые компиляторе оптимизации. Но они применяются достаточно редко. У приватного наслдеования есть определенные плюсы, но они находят довольно редкое примененние. «Мне больше нравится» явное включение, с ним все проще уследить. Мы должны четко понимать, почему мы наследуем так, а не иначе.

Protected наследование:

public | => protected — все public и protected становятся protected
private |

```
class Apple: protected Fruit {
public:
        string getName( );
        …
};

string gitName( )
{
        return Fruit::getName( );
} // keyword using, let's see

class Apple: protected Fruit {
public:
        using Fruit::getName( ); //refers to original class
        …
};
```

Множественное наследование:

Швейцарский ножик <= сразу все: ножик, ложка, вилка и т.д.

```
class SwissKnife: public Knife; public spoon {
        Swiss knife state;
};

class Equipment {
        MaterialType type;
        int date;
        virtual getInfo( ); //правильно сделать его virtual, хотя где-то это не обязательно
}; // если метод виртуальный, то он доминирует при вызове

class Knife: public Equipment {
        double length;
        gerInfo( );
};

class Spoon: public Equipment {
        double R; //curving radius
        getInfo( );
};

SwissKnife ( MaterialType mType, int date, int length, double R, SwissKnife state);
:Knife(mType, date, length);
 Spoon(mType, date, R);
 State(state); // явное переполнение, как исправить? ->

SwissKnife Sk(…);
Spoon &sp = Sk; // восходящее приведение типов
sp.getMaterial( ); // check this via book

class worker { // пример из книги Стивена Прата
private:
        string name;
};

class waiter: public worker {
```

```
private:
        int ind;
};

class Singer: public worker {
private:
        int tune;
};
```

Worker -> Waiter -> SingerWaiter
-> Singer ->

Проблемы множественного наследования:
1. Избыточное использование памяти
2. Наследование одинаковых методов от разных классов
3. Восходящее приведение типов становится не совсем очевидным

Пример:

```
Equipment & Eq = Sk; // which method to choose?
Equipment & Eq = Knife(Sk); // here we define the way we go
                = Spoon(Sk);
```

Можно сделать отдельный объект, для этого добавили виртуальные базовые классы. Чтобы Equipment им стал для Knife и Spoon мы должны добавить virtual.

```
class Knife: virtual public Equipment { … };
class Spoon: virtual public Equipment { … }; // тогда память имеет структуру рис.2

Equipment & Eq = Sk; // теперь работает
Knife & Kn = Sk;
Spoon & Sp = Sk;

SwissKnife( Equipment & Eq, double length, double R, SwissState state)
: Equipment ( Eq ),
  Knife ( Eq, length ),
  Spoon ( Eq, R ),
  State ( state ) { … }
```

Stack — here we got interface LIFO and an object it works with. And you can take legacy from both of them … ?

Deck — LIFO and FIFO

```
getInfo( ) {
        Equipment::getInfo( );
        cout << length;
}
```

void getInfo … the information had been erased :(

```
void getInfo( ) {
        Knife::getInfo( ); // нужно изменять, уже к существующей функции еще что-то
        Spoon::getInfo( ); // написали, но при множественном наследовании это не
}                          // очень работает, так что нужно придумать какие-то методы.

… getData( ); // получаем в отдельности
Equipment::getData( );
Knife::getData( );
Spoon::getData( );
```

A -> (virtual) B, C -> F // в F будет в итоге 3 объекта класса A
  -> D, E -> // котлер, язык from Russia

Пользоваться ли множественным наследованием? Используйте common sense.