

Lab 3. Part II. Semantic analyzer
for cool.

Mipt (Ilab), 4.12.2018

Еще раз о C++

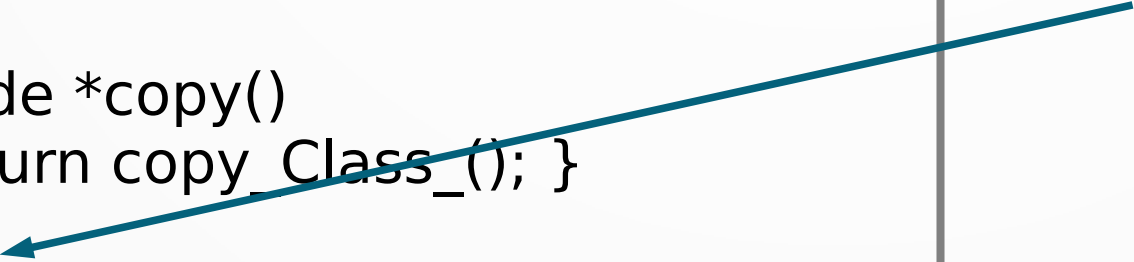
Виртуальные методы

```
// define simple phylum - Class_
typedef class Class__class *Class_;


class Class__class : public tree_node {
public:
    tree_node *copy()
        { return copy_Class_(); }

    virtual Class_ copy_Class_() = 0;
#   ifdef Class__EXTRAS
        Class__EXTRAS
#   endif
};
```

Метод виртуальный
(переопределяется в потомках)




Реализации нет.
То есть весь класс Class__class -
виртуальный



Еще раз о C++

Шаблоны

Параметры шаблона



```
template <class SYM, class DAT>
class SymbolTable
{
    typedef SymtabEntry<SYM, DAT> ScopeEntry;

    ScopeEntry *addid(SYM s, DAT *i)

    DAT * lookup(SYM s)

    DAT *probe(SYM s);

    void dump();
}
```

В классе могут использоваться как классы переменных, аргументы функций, возвращаемые значения

То есть в место любого типа

Так же есть возможность использовать шаблонные параметры как значения переменных (константы). Однако, в текущем проекте такого нет.

ГОЛОВНОЙ МЕТОД.

```
void program_class::semant()
{
    initialize_constants();

    /* ClassTable constructor
       may do some semantic analysis */

    ClassTable *classtable =
        new ClassTable(classes);

    if (classtable->errors()) {
        cerr << "Compilation halted due ";
        cerr << "to static semantic errors.";
        cerr << endl;
        exit(1);
    }
}
```

classes объект типа **Classes**

(ast_parse.cpp + 198)

Тип **Classes** в свою очередь является указателем на объект типа **Classes_class**

typedef Classes_class *Classes;
(cool-tree.h + 104)

Который в свою очередь является нодой типа **Class**

typedef list_node<Class_> Classes_class;
(cool-tree.h + 103)
typedef class Class__class *Class_;
(cool-tree.h + 32)

Класс **Class__class** - наследник tree_node
(cool-tree.h + 34)

Объект **classes** умеет

->first()

->next(i)

->more(i) и многое другое!

Class, class, cLaSs ...

Виртуальный

class **Class__class** : public tree_node ; Описание одного класса

typedef class Class__class ***Class_**;

Переопределение типов –
указатель на описание класса

Виртуальный

typedef list_node<Class_> **Classes_class**;

Параметризация шаблона –
список из указателей на классы
(в действительности узел дерева)

typedef Classes_class ***Classes**;

Переопределение типа –
указатель на описание класса

class class__class : public Class__class

Настоящий узел дерева – с
переопределенными виртуальными ф-ями

Вход в программу

semant-phase.cc

```
int main(int argc, char *argv[]) {
```

```
    handle_flags(argc,argv);
```

```
    ast_yyparse();
```

```
    ast_root->semant();
```

```
    ast_root->dump_with_types(cout,0);
```

```
}
```

Парсинг входящего
дерева

Вызов функции с
предыдущего слайда

Полный дамп дерева

Деревья

list_node< **Elem** > Class Template Reference

Public Member Functions

tree_node*	copy ()	
Elem	nth (int n)	← Получение элемента
int	first ()	
int	next (int n)	
int	more (int n)	
virtual list_node< Elem >*	copy_list ()=0	
virtual int	len ()=0	
virtual Elem	nth_length (int n, int &len)=0	

Inherited from tree_node

virtual void int	dump (ostream &stream, int n)=0
int	get_line_number ()
tree_node*	set (tree_node *)

Static Public Member Functions

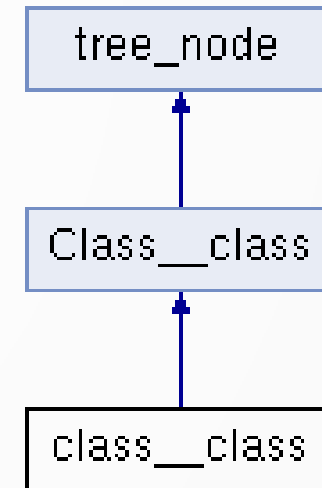
static list_node< Elem > *	nil ()	
static list_node< Elem > *	single (Elem)	
static list_node< Elem > *	append (list_node< Elem > *l1, list_node< Elem > *l2)	

Определены и могут вызываться для всех объектов данного класса (даже напрямую из класса, без создания объекта)

Пример логики работы.

Предположим, мы хотим распечатать все фичи класса.

```
class class__class : public Class__class {
protected:
    Symbol name;
    Symbol parent;
    Features features;
    Symbol filename;
Public:
    class__class(Symbol, Symbol, Features, Symbol) {...}
    Class__copy_Class_();
    void dump(ostream& stream, int n);
}
```



```
ClassTable::ClassTable(Classes classes) : semant_errors(0) , error_stream(cerr)
{
    std::cout << "Print Class table \n";
    for(int i = classes->first(); classes->more(i); i = classes->next(i)) {
        Class__class_ = classes->nth(i);
        //class_->dump_with_types(std::cout, 2); // <- it is work !!!
        for ( int i = class_->features->first();class_->features->more( i);i = class_->features->next( i))
        {
            class_->features->nth( i)->dump_with_types(std::cout, 2); // ???
        }
    }
}
```


Увы, так не работает.

error: cannot initialize a variable of type 'class__class *' with an rvalue of type 'Class__class *'

Для того, чтобы получить доступ к данным объектов – необходимо создать виртуальные методы в их родителях.

То есть:

- Идем в Class__class (cool-tree.h), добавляем туда виртуальных методов для работы с элементами.
- Идем в class__class, конкретизируем эти методы.
- При обходе указателей на Class__class вызываем виртуальные методы, которые реализованы в class__class

КТО ТАКИЕ ТИПЫ?

В cool любой тип – это объект класса.

Но их пока нет. Нужно их сделать.

До этого момента все типы - это только строки. Однако где-то надо их хранить. Например могут храниться например в
Symbol name;

Для этого приведен список в начале semant.cc:

```
class method_class : public Feature_class {  
  
protected:  
    Symbol name;  
    Formals formals;  
    Symbol return_type;  
    Expression expr;  
    ...  
}
```

```
static Symbol  
    Bool,  
    Int,  
    IO,  
    Main,  
    Object,  
    out_int,  
    out_string,  
    prim_slot,  
    self,  
    SELF_TYPE,  
    Str,  
    str_field,  
    substr,  
    main_meth,  
    No_class
```

Области видимости (symtab.h)

Для удобства работы с областями видимости существует шаблон класса:

```
template <class SYM, class DAT> class SymbolTable
```

SYM – символ **DAT** – данные

Он как раз реализован как список списков. Его публичные методы (и методы с прошлых слайдов) :

enterscope()

enter_scope()

exitscope()

exit_scope()

addid(SYM s, DAT *i)

add_symbol(x)

lookup(SYM s)

find_symbol(x)

probe(SYM s)

check_scope(x)

dump()

Пример реализации проверки типов.

```
void cond_class::semant()  
{  
    pred->semant();  
    then_exp->semant();  
    else_exp->semant();  
    if (pred->get_type() == Bool) {  
        type = curr_classtable->lub(then_exp->get_type(), else_exp->get_type());  
    } else {  
        curr_classtable->semant_error(curr_class);  
        cerr << "Predicate of conditional is not of type Bool" << endl;  
        type = Object;  
    }  
}
```

$$\frac{\begin{array}{c} O \vdash e_0 : Bool \\ O \vdash e_1 : T_1 \\ O \vdash e_2 : T_2 \end{array}}{O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)} \quad [If - Then - Else]$$

Проверку lub нужно
писать самому.

Есть дефайн Expression_EXTRAS - они определены у
родителя всех выражений

```
#define Expression_EXTRAS \
Symbol type; \
virtual Symbol get_type(); \
Expression set_type(Symbol s) { type = s; return this; } \
virtual void dump_with_types(ostream&,int) = 0; \
void dump_type(ostream&, int); \
SEMANT_VIRTUAL_FUNCT \
Expression_class() { type = (Symbol) NULL; }
```

```
#define SEMANT_VIRTUAL_FUNCT \
virtual void semant(class_list_type*,attr_list_type*,method_list_type*) = 0;
```

ClassTable::install_basic_classes

В cool есть predefined классы
(cool_manual.pdf 8. Basic Classes):

Object, IO, Int, String, Bool

Все они должны быть определены до начала семантического анализа.

При конструировании – они получают имена, определенные в Symbol, при этом используется метод, а-ля "Фабрика" (метод, возвращающий новый объект класса) + вызываются методы по созданию feature.

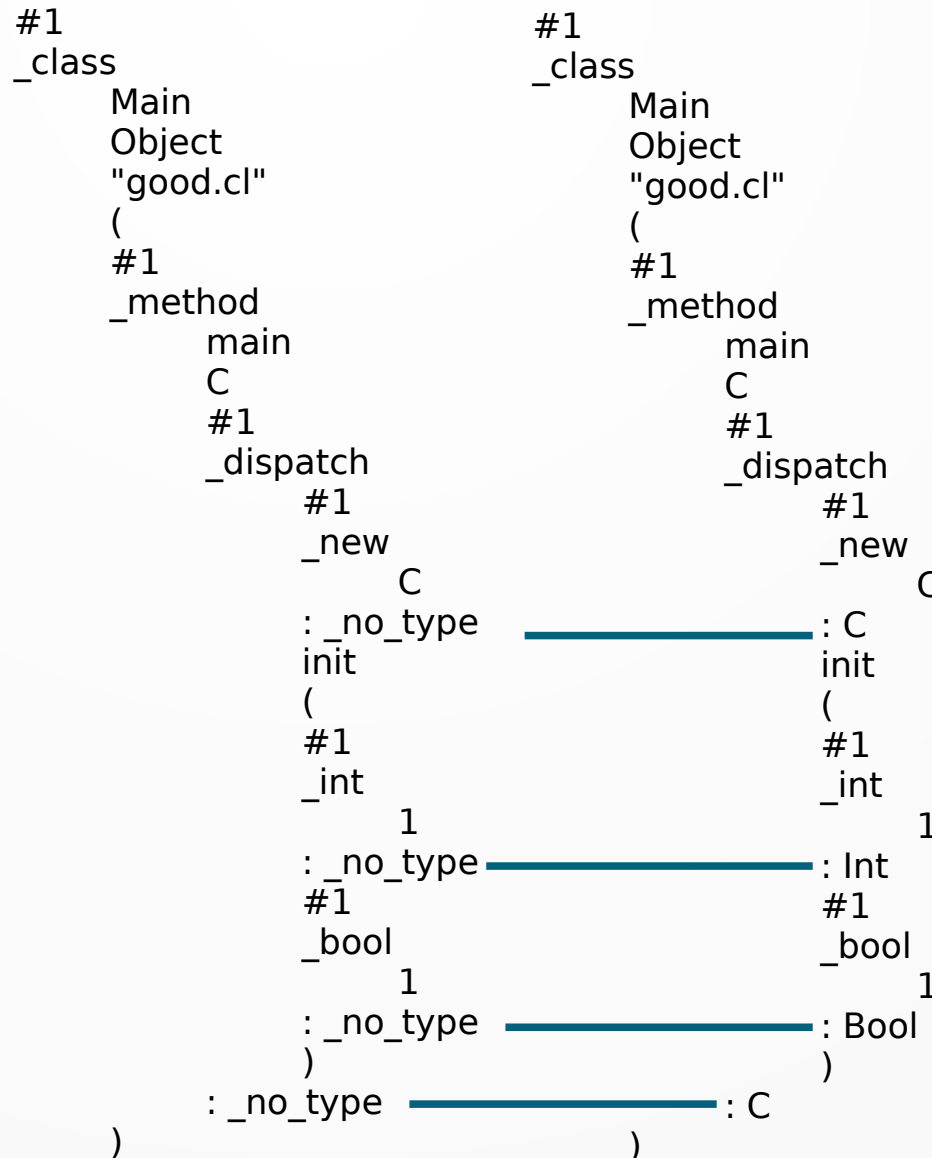
```
Class_IO_class =  
    class_(IO, Object,  
        append_Features( append_Features( append_Features(  
            single_Features(method(out_string,  
                single_Formals(formal(arg, Str)),  
                SELF_TYPE, no_expr()))),  
            single_Features(method(out_int,  
                single_Formals(formal(arg, Int)),  
                SELF_TYPE, no_expr()))),  
            single_Features(method(in_string,  
                nil_Formals(), Str, no_expr()))),  
            single_Features(method(in_int,  
                nil_Formals(), Int, no_expr()))),  
        filename);
```

Как это выглядит на тестах

```
class C {  
  a : Int;  
  b : Bool;  
  init(x : Int, y : Bool) : C {  
    {  
      a <- x;  
      b <- y;  
      self;  
    }  
  };  
};
```

```
Class Main {  
  main():C {  
    (new C).init(1,true)  
  };  
};
```

```
#1  
_class  
Main  
Object  
"good.cl"  
(  
  #1  
  _method  
  main  
  C  
  #1  
  _dispatch  
  #1  
  _new  
  C  
  : _no_type  
  init  
  (  
    #1  
    _int  
    1  
    : _no_type  
    #1  
    _bool  
    1  
    : _no_type  
    )  
  : _no_type  
  )  
)
```



```
#1  
_class  
Main  
Object  
"good.cl"  
(  
  #1  
  _method  
  main  
  C  
  #1  
  _dispatch  
  #1  
  _new  
  C  
  : C  
  init  
  (  
    #1  
    _int  
    1  
    : Int  
    #1  
    _bool  
    1  
    : Bool  
    )  
  : C  
  )  
)
```

Создаваемый
класс:

new__class

int_const_class

bool_const_class
dispatch_class

Пример логики работы #2.

В Class__class добавляем виртуальный метод:

-> virtual void my_dump (ostream &stream, int n) = 0;

В class__class - добавляем реализацию (**cool-tree.h**):

```
void my_dump(ostream& stream, int n)
{
    for (    int i = features->first();
            features->more( i);
            i = class_->features->next( i))
    {
        std::cout << "Dump " <<
            features->nth( i)->get_line_number() << "line\n";
        features->nth( i)->dump_with_types(std::cout, 2);
    }
};
```

И вызываем в (**semant.cc**):

```
ClassTable::ClassTable(Classes classes) : semant_errors(0) , error_stream(cerr) {
    std::cout << "Init Class table";
    for(int i = classes->first(); classes->more(i); i = classes->next(i)) {
        Class__class* class_ = classes->nth(i);
        class_->my_dump(std::cout, 2);
    }
}
```


Маленький пример.

Рассмотрим пример (cool-manual.pdf 4.1)

```
class Silly {  
    copy() : SELF_TYPE { self };  
};  
class Sally inherits Silly { };  
class Main {  
    x : Sally <- (new Sally).copy();  
    main() : Sally { x };  
};
```

Какого типа будет x?

$$\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T'_0 = \begin{cases} C & \text{if } T_0 = \text{SELF_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T'_0, f) = (T'_1, \dots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T'_{n+1} = \text{SELF_TYPE} \\ T'_{n+1} & \text{otherwise} \end{cases} \\ \hline O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_{n+1} \end{array}$$

Здесь SELF_TYPE просто подменяется на тип класса.

В раз для работы с классами нужно:

1. Создать список для хранения объявленных имен классов.
2. Создать список для хранения описания методов и переменных классов.
3. Уметь строить `lub(lca)` от двух классов.
4. Заполнить списки.

Для работы с определениями(expression):

1. Уметь кидать ошибки.
2. Уметь получать типы.
3. Уметь выставлять типы.

Необходимо

- 1) Обходить дерево
- 2) Уметь получать и выставлять типы для объектов
(рекурентный гетер + гетеры из таблиц классов, методов и score).
- 3) Проверять типы на основании правил (cool-manual.pdf : 12.1)