

# Compilers Principles Lab5 Report

## *Language Feature Extension*

Yurun Yuan

CS, USTC

January 21, 2022

# Outline

- 1 Type System Extension
  - Pointers and Arrays
  - Structures
- 2 Operator Extension
  - Context Free Grammar of Expressions
  - Operations on Structures
- 3 Classes and Templates
  - Non-Static Member Functions
  - Operator Overloading
  - Class Templates

# Outline

- 1 Type System Extension
  - Pointers and Arrays
  - Structures
- 2 Operator Extension
  - Context Free Grammar of Expressions
  - Operations on Structures
- 3 Classes and Templates
  - Non-Static Member Functions
  - Operator Overloading
  - Class Templates

# Declarations of Variables

- Pointers

**int** \*ptr;

- Function

**int** func(**int**);

- Arrays

**int** array [42];

- Structures

**struct** S {...};

**struct** S s;

**struct** S {...} s;

**struct** {...} anonymous;

# Declaration Grammar

## Declaration grammar

declaration  $\rightarrow$  type-specifier declarator ;

## Example

type-specifier

int

\* a[2] ;

declarator

# Declaration Grammar

## Declaration grammar

declaration  $\rightarrow$  type-specifier declarator ;

- Pointer declarators  
`int * ptr ;`
- Function declarators  
`int func(float) ;`
- Array declarators  
`int array[42] ;`

# Crux: declarators can get mixed up

## Example

```
int (* a [2])( int );
```

Declarators	Declarator type	Type expression
( * a [2] )( <b>int</b> )	Function	<b>int</b>
* a [2]	Pointer	<b>int</b> → <b>int</b>
a [2]	Array	pointer( <b>int</b> → <b>int</b> )
a		array(2, pointer( <b>int</b> → <b>int</b> ))

# Trial 1: a naive solution

## Example

```
int (* a [2])( int );
```

declarator	→	* declarator	<i>Pointer declarator</i>
		declarator ( type, ... )	<i>Function declarator</i>
		declarator [ int ]	<i>Array declarator</i>
		( declarator )	
		ID	

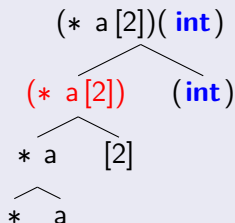
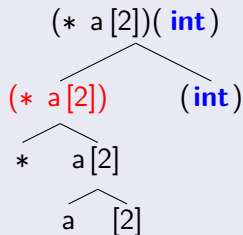


# Trial 1: a naive solution

## Example

```
int (* a [2])( int );
```

## Problem: ambiguity



## Trial 2: eliminating ambiguity

- Parentheses and identifiers

$$\begin{array}{l} \text{factor} \rightarrow \mathbf{ID} \\ \quad \quad | \text{ ( decl )} \end{array}$$

- Function calls and array subscripting

$$\begin{array}{l} \text{decl-1} \rightarrow \text{decl-1}(\text{type}, \dots) \\ \quad \quad | \text{ decl-1}[\mathbf{int}] \\ \quad \quad | \text{ factor} \end{array}$$

- Dereference

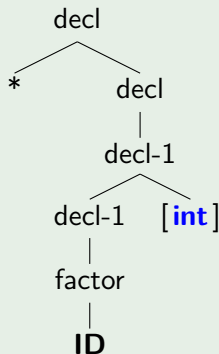
$$\begin{array}{l} \text{decl} \rightarrow * \text{ decl} \\ \quad \quad | \text{ decl-1} \end{array}$$

Preced	Operation
0	Parenthesis ()
1	Call () subscript []
2	Dereference *

# Trial 2: eliminating ambiguity

## Example

```
int * array [2] ;
```



- Parentheses and identifiers

$$\text{factor} \rightarrow \mathbf{ID}$$
$$\quad \quad \quad | \quad ( \text{ decl } )$$

- Function calls and array subscripting

$$\text{decl-1} \rightarrow \text{decl-1}(\text{type}, \dots)$$
$$\quad \quad \quad | \quad \text{decl-1}[\mathbf{int}]$$
$$\quad \quad \quad | \quad \text{factor}$$

- Dereference

$$\text{decl} \rightarrow * \text{ decl}$$
$$\quad \quad \quad | \quad \text{decl-1}$$

# Declarations of Structures

- Structure definition  
**struct** S {...};
- Variable definition  
**struct** S s;
- Structure and variable definition  
**struct** S {...} s;
- Anonymous structure  
**struct** {...} anonymous;

# Extension of the declaration grammar

## Declaration grammar

declaration  $\rightarrow$  type-specifier declarator ;

Declaration	Type specifier	Declarator
<b>struct</b> S { ... } ;	<b>struct</b> S { ... }	$\emptyset$
<b>struct</b> S s ;	<b>struct</b> S	s
<b>struct</b> S { ... } s ;	<b>struct</b> S { ... }	s
<b>struct</b> { ... } anonymous ;	<b>struct</b> { ... }	anonymous

# Extension of the declaration grammar

## Declaration grammar

declaration  $\rightarrow$  type-specifier declarator ;

type-specifier  $\rightarrow$  ...  
| **struct** ID {definitions}  
| **struct** {definitions}  
| **struct** ID

declarator  $\rightarrow$  ...  
|  $\epsilon$

# Self reference

## Example

A naive implementation of the node of a forwarding linked list

```
struct list_node{  
    int elem;  
    struct list_node* next;  
};
```

The identifier `struct list_node` is visible within the definition of the structure, although `struct list_node` is an incomplete type.

# Self reference

## Example

A naive implementation of the node of a forwarding linked list

```
struct list_node{  
    int elem;  
    struct list_node* next;  
};
```

To render the identifier `struct list_node` available, construct an empty structure before parsing the body of the structure definition, and complete the type after the definition parsing is done.



# Outline

- 1 Type System Extension
  - Pointers and Arrays
  - Structures
- 2 Operator Extension
  - Context Free Grammar of Expressions
  - Operations on Structures
- 3 Classes and Templates
  - Non-Static Member Functions
  - Operator Overloading
  - Class Templates

# Supported operators

- Arithmetic operators  
+, -, \*, /
- Relational operators  
>, <, >=, <=, ==, !=
- Assignment operator  
=
- Array subscript  
array[index]
- Pointer dereference  
\* ptr
- Address of  
& lvalue
- Member access  
var.member
- Function call  
callable (params)

# Precedence and CFG

## 0 Parenthesis ()

$\text{expr-0} \rightarrow$  **ID**  
| **Integer-literal**  
| **Float-literal**  
| ( expr )

## 1 Function call, array subscript, member access

$\text{expr-1} \rightarrow \text{expr-1 ( args )}$   
|  $\text{expr-1 [ expr ]}$   
|  $\text{expr-1 . ID}$   
|  $\text{expr-0}$

# Precedence and CFG

## ② Dereference, address of

$$\begin{aligned} \text{expr-2} &\rightarrow * \text{expr-2} \\ &| \& \text{expr-2} \\ &| \text{expr-1} \end{aligned}$$

## ③ Multiplication, division

$$\begin{aligned} \text{expr-3} &\rightarrow \text{expr-3} \mathbf{MulOp} \text{expr-2} \\ &| \text{expr-2} \end{aligned}$$

## ④ Addition subtraction

$$\begin{aligned} \text{expr-4} &\rightarrow \text{expr-4} \mathbf{AddOp} \text{expr-3} \\ &| \text{expr-3} \end{aligned}$$

# Precedence and CFG

## 5 Relational operations

$$\begin{array}{l} \text{expr-5} \rightarrow \text{expr-5} \textbf{RelOp} \text{expr-4} \\ \quad \quad \quad | \text{expr-4} \end{array}$$

## 6 Assignment

$$\begin{array}{l} \text{expr} \rightarrow \text{expr-5} = \text{expr} \\ \quad \quad \quad | \text{expr-5} \end{array}$$

### Note

The assignment operator is right associated, so the production is  $\text{expr} \rightarrow \text{expr-5} = \text{expr}$  instead of  $\text{expr} \rightarrow \text{expr} = \text{expr-5}$ .

# Problems with operations on structures

- Assignment to structures  
**struct** S s = t;
- Structures as parameters  
**void** func(**struct** S ){...};
- Structures as return values  
**struct** S func () {...};

# Assignment to structures

- Source code

```
struct S
{
    int member1;
    float member2;
};

s = t;
```

- Transformed code

```
struct S{
    int member1;
    float member2;
};

s.member1 = t.member1;
s.member2 = t.member2;
```

# Structures as parameters

- Source code

```
void func(struct S s)
{
    ...
}

int main()
{
    func(t);
}
```

- Transformed code

```
void func(struct S *ptr)
{
    struct S s = *ptr;
}

int main()
{
    func(&t);
}
```



# Structures as return values

- Source code

```
struct S func()  
{  
    ...  
    return s;  
}  
  
int main()  
{  
    func();  
}
```

- Transformed code

```
void func(struct S* ret_ptr)  
{  
    ...  
    *ret_ptr = ret_value;  
}  
  
int main() {  
    struct S ret_value;  
    func(&ret_value);  
}
```

# Outline

- 1 Type System Extension
  - Pointers and Arrays
  - Structures
- 2 Operator Extension
  - Context Free Grammar of Expressions
  - Operations on Structures
- 3 Classes and Templates
  - Non-Static Member Functions
  - Operator Overloading
  - Class Templates

# Functions declared within a structure

## Example

```
struct S
{
    ...
    void func(){}
};
```

```
int main()
{
    struct S s;
    s.func();
}
```

# Transformation of member functions

- Source code

```
struct S
{
    ...
    void func(){}
};

int main()
{
    struct S s;
    s.func();
}
```

- Transformed code

```
struct S
{
    ...
};
void S::func(struct S* this){}

int main()
{
    struct S s;
    S::func(&s);
}
```

# Overload an operator

```
struct S {  
    struct S operator+(struct S rhs)  
    {...}  
};  
  
int main()  
{  
    struct S s;  
    struct S t;  
    ...  
    s + t; // S::operator+ is called  
}
```

# Transform the overloading function

- Source code

```
struct S
{
    S operator+(S)
    {...}
};
```

```
int main() {
    struct S s;
    struct S t;
    ...
    s + t;
}
```

- Transformed code

```
void S::operator+(
    struct S* ret_ptr,
    struct S* this
    struct S* rhs_ptr){
    struct S rhs = *rhs_ptr;
    ...
    *ret_ptr = ret_value;
}
```

```
int main(){
    struct S s; struct S t;
    ...
    struct S ret_value;
    S::operator+(&ret_value, &s, &t);
}
```

# Generic programming

## Example

```
template<typename T>
struct stack{
    T top();
    void push(T);
};
```

```
int main()
{
    stack<int> a;
    stack<string> b;
    a.push(42);
    b.push("42");
}
```

# CFG for class templates

template-definition  $\rightarrow$  **template** < **typename ID**, ...>  
struct-definition

type-specifier  $\rightarrow$  ...  
| **typename ID**  
| **struct ID** <declaration,...>

## Note

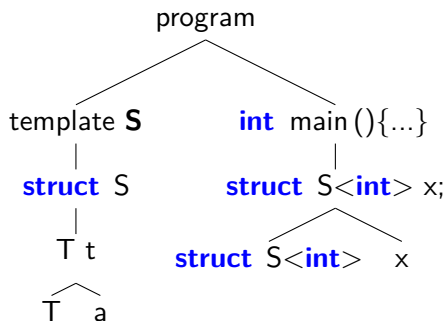
Due to the limitation of CFG, in the specification of a class template, the keyword *typename* is required before a template parameter.



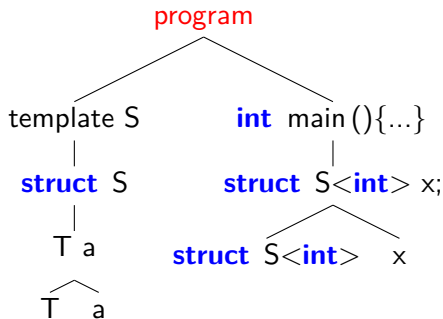
# Translation process of a class template

```
template<typename T>
struct S{
    typename T a;
};

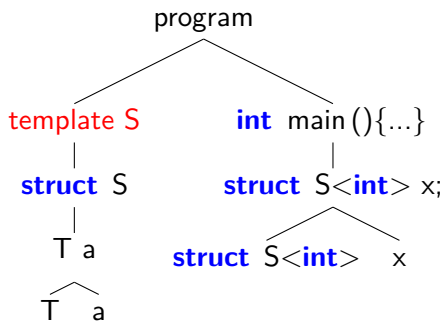
int main(){
    struct S<int> x;
}
```



# Translation process of a class template

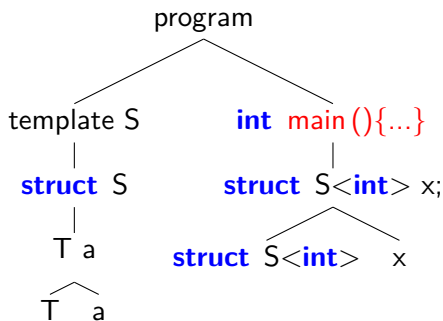


# Translation process of a class template



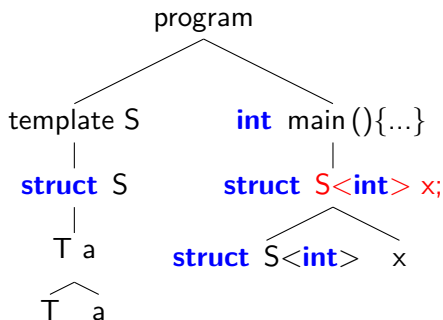
- Record the pointer of node **template S** in AST, do not analyse subtrees.

# Translation process of a class template



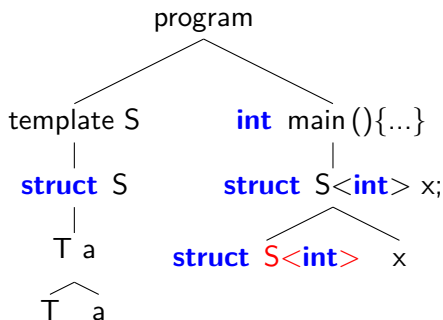
- Record the pointer of node template S in AST, do not analyse subtrees.

# Translation process of a class template



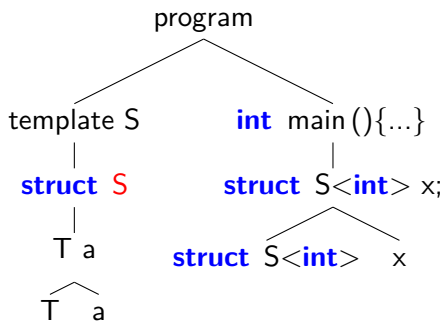
- Record the pointer of node template S in AST, do not analyse subtrees.

# Translation process of a class template



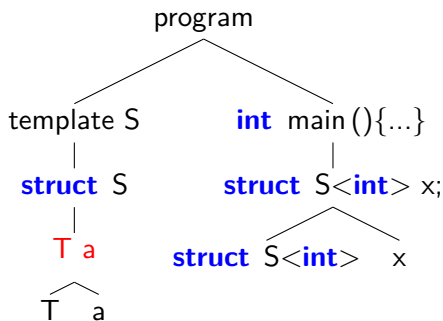
- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<int>$  does not exist. Map parameter  $T$  to  $int$ , analyse class template

# Translation process of a class template



- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- **Construct structure  $S<\text{int}>$**

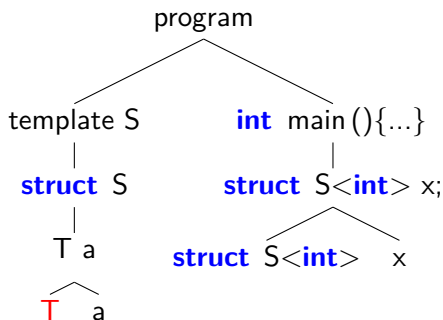
# Translation process of a class template



- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- Construct structure  $S<\text{int}>$

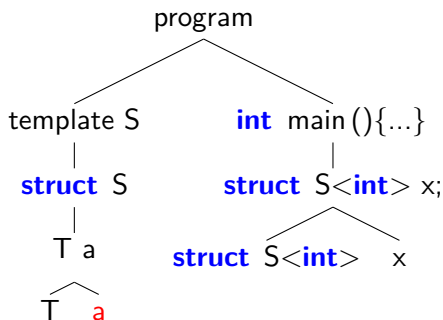


# Translation process of a class template



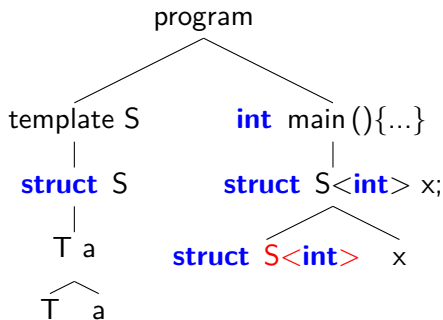
- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- Construct structure  $S<\text{int}>$
- **Use mapping to replace  $T$  with  $\text{int}$**

# Translation process of a class template



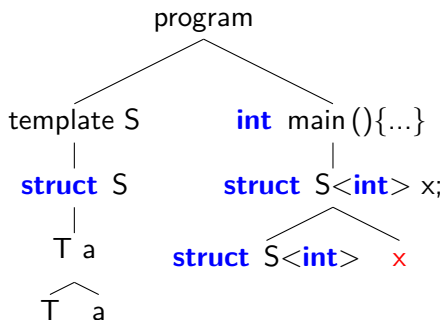
- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- Construct structure  $S<\text{int}>$
- Use mapping to replace  $T$  with  $\text{int}$

# Translation process of a class template



- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- Construct structure  $S<\text{int}>$
- Use mapping to replace  $T$  with  $\text{int}$
- $S<\text{int}>$  now is a complete type

# Translation process of a class template



- Record the pointer of node template  $S$  in AST, do not analyse subtrees.
- $S<\text{int}>$  does not exist. Map parameter  $T$  to  $\text{int}$ , analyse class template
- Construct structure  $S<\text{int}>$
- Use mapping to replace  $T$  with  $\text{int}$
- $S<\text{int}>$  now is a complete type
- **Declare a variable of type  $S<\text{int}>$**

*Thank You.*