

Faculty of Information Technology
Damascus University
Department of Software Engineering and Information Systems

Compiler Project
MiniC++

prepared and implemented by:
Yasser Almohammad

Under Supervision of
PH.D. Kalel Alajami

2006

White paper

Table of Contents:

[1]	Introduction	[6-9]
1-1	Lexical Analysis Phase	[8]
1-2	Syntax Analysis	[8]
1-3	Semantic Analysis	[8]
1-4	Code Optimization	[8]
1-5	Code Generation	[9]
[2]	Lexical Analysis:	[9-14]
2-1	Include File Handling	[10-12]
	INCLUDE STATE	[11]
2-2	Class Type – Variable Type discrimination	[12]
2-3	Comments handling	[13]
2-4	Line-column tracking	[13-14]
[3]	Syntax Analysis:	[15-22]
3-1	Conflict resolving approach	[15]
3-2	Sample rules	[16-19]
3-3	Shift/Reduce Conflicts and resolving:	[19-20]
3-4	Error recovery	[20-22]
3-5	Location Tracking	[22]
[4]	Semantic Analysis:	[23-50]
4-1	Symbol Table management and creation	[23-25]
4-1-1	What information do we store inside this symbol table? [23]	
4-1-2	Symbol Table structure	[23-24]
4-1-3	Symbol table Content Sample	[25]
4-1-4	The hash function	[25]
4-2	The Abstract Syntax Tree	[26]
4-2-1	Node Structure	[26]
4-3	Type Checking & AST Nodes	[27]
4-3-1	first node is the prog_decls	[27]
4-3-2	class definition node	[27-28]
4-3-3	class body node	[28]
4-3-4	class body statement node	[28]
4-3-5	access specification node	[29]
4-3-6	class constructor declaration , Function Overloading, Function Signature coding	[29-30]
4-3-7	class constructor node	[30-31]
4-3-8	class destructor declaration node	[31]
4-3-9	class declaration node	[31-32]
4-3-10	function declaration node	[32]
4-3-11	function definition node	[33]
4-3-12	class function definition node	[33]
4-3-13	statements node	[34]
4-3-14	delete statement node	[34]
4-3-15	cin statement node	[34]
4-3-16	cout statement node	[34]

4-3-17	block statement	[35]
4-3-18	variable declaration nodes	[35]
4-3-19	array dims node, array information storage, and array initialization list handling	[36-39]
4-3-20	class static member initialization node	[39]
4-4-21	class destructor node	[40]
4-3-22	stand alone block statement	[40]
4-4-23	jump statements	[40]
4-3-24	for statement	[41]
4-3-25	if statement	[41-42]
4-4-27	assignment node	[42]
4-3-28	bracket expression	[43]
4-3-29	constant expressions	[43]
4-3-30	a variable expression node	[43-44]
4-3-31	array expression node	[44]
4-3-32	array of objects expression	[44-45]
4-3-33	array of object pointer	[45]
4-3-34	variable expression node	[45]
4-3-35	this expression node	[45]
4-3-36	class variable expression	[46]
4-3-37	class static members	[46]
4-3-38	object pointer expression node	[46]
4-3-39	this pointer expression node	[46]
4-3-40	unary expression nodes	[47]
4-3-41	binary expression nodes	[48]
4-3-42	mathematical operations expression nodes	[48]
4-3-43	logical operations expression nodes	[48]
4-3-44	allocate-new expression	[49]
4-3-45	cast expression node	[49]
4-3-46	procedure call nodes	[49-50]
[5]	Code Optimization:	[51-56]
5-1	Eliminating dangling expressions	[51]
5-2	deleting unused variables	[52]
5-3	deterministic if, if else statements	[52-53]
5-4	deterministic loop statements	[53]
5-5	constant expressions calculations	[53-54]
5-6	sequence of ANDs-ORs cut	[54-55]
5-7	loop expansion (or loop unrolling)	[55-56]
[6]	Code Generation:	[57-81]
6-1	Code initialization	[58]
6-2	Variable Storage: and retrieval introduction	[58-59]
6-3	complex variable initialization	[59]
6-4	variable retrieval	[59]
6-5	AST nodes Code generation details	[60-81]
6-5-1	program declaration nodes	[60]
6-5-2	class definition node	[60-61]

6-5-3	function definition code:[61-63]
6-5-3-1	Unique labeling (naming) generation:	[61]
6-5-4	class function definition code:[63]
6-5-5	class constructor definition code:[63]
6-5-6	class destructor definition code:[63]
6-5-7	block statements:[63-64]
6-5-8	delete statement:[65]
6-5-9	expression statements code generation[65]
6-6-10	procedure call statement code[65-66]
6-5-11	unary operation statement generation[67]
6-5-12	if statement code:[67-68]
6-5-13	for statement code:[68-69]
6-5-14	while statement code:[69-70]
6-5-15	jump statement code:[70-71]
6-5-16	assignment and variable storage retrieval code:	[72-74]
6-5-17	Object Instantiation:[75]
6-5-18	Generating math operation code[76]
6-5-19	Generating Cast Balancing Code:[76]
6-5-20	unary operations:[76]
6-5-21	logical operation code:[76-78]
6-5-22	The AND-OR tree cut:[78-79]
6-5-23	cout statement code[80]
6-5-34	cin statement code:[80-81]
7-	Real World Examples:[82-94]
7-1	Example1: Factorial[82-83]
7-2	Example 2 : Lined List[84-86]
7-3	Example 3 : multi-level inheritance application[87-89]
7-3	Example 3 : Object Oriented Stack application[90-93]
8-	Supplement:	
	Simplified Class Diagram[94-94]

Table of Contents Ends here.

1- Introduction:

A compiler is a program, that translates a high level language context programs into low level language that could be executed on a target machine [source – target]
And because this process is very complex; hence, from the logical as well as an implementation point of view, it is customary to partition the compilation process into several phases, some can be completely separated from others and some are well connected with each other.

What we are going to do here, is build a compiler for a language like C++ which we'll call it MiniC++.

The program we make, takes a file written in the previously mentioned language, does what it does, to check and find out as much as possible all kinds of syntax and semantic errors then if the program is fine, we generate a target VM machine code file to be executed.

So the process can be expressed as:



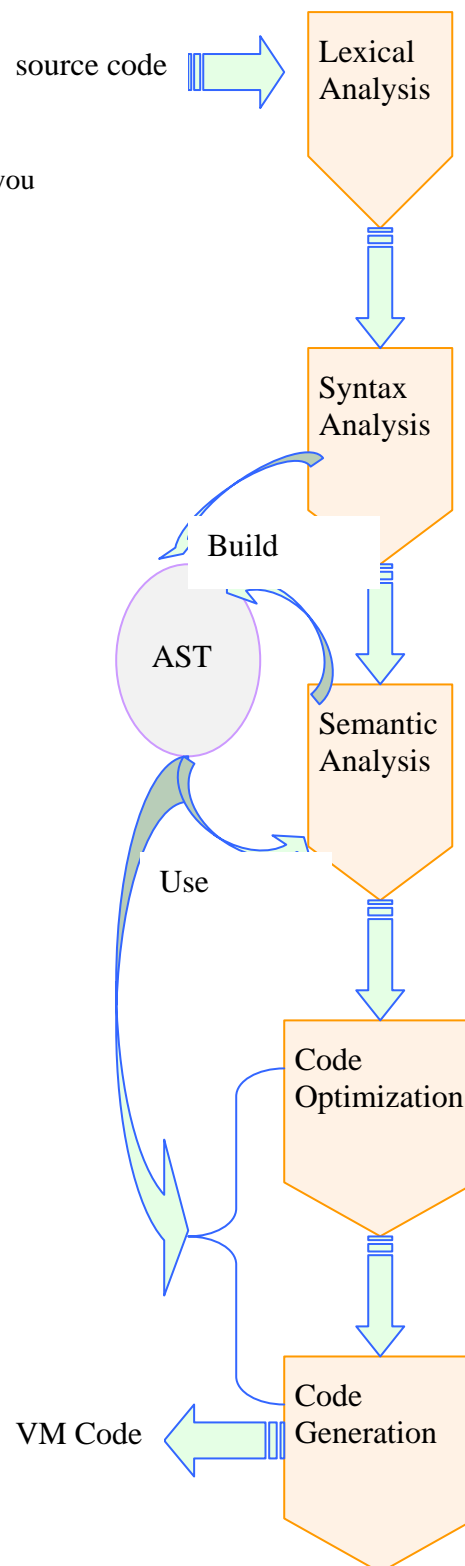
We shall make a brief description of compiler design phases, and then begin more depth description of each phase, , implementation steps also will be described, and finally tests are to be included.

The phases into which a fully functional compiler to be achieved could be described as:

Though phases could interlace with each Other How ever separation of phases could Put The focus to make one phase work fine Before moving to the next phase. However, when a phase is designed, the next phase is taken into consideration for example: you can't design your AST without taking Code Generation into consideration.

Some parts of compiler design could be distributed between Others for example:

Code Optimization could partially be Done during building the Abstract Syntax Tree AST also during The Code Generation, or even after it's done, so there is no actual Restriction on how you choose your work to be done to get this Functional compiler.



1-1 Lexical Analysis Phase:

In the lexical analysis phase, the compiler scans the characters of the source program, one character at a time. Whenever it gets a sufficient number of characters to constitute a token of the specified language, it outputs that token. In order to perform this task, the lexical analyzer must know the keywords, identifiers, operators, delimiters, and punctuation symbols of the language to be implemented.

Some characters are ignored like: new lines, spaces, tabs...

Also comments are identified and delimited at this stage, since there is no need to trouble next stages with it.

Since we use LEX to do this task, our job in this stage becomes only to create the rules necessary to recognize this language, these rules are regular expressions, and this LEX creates all necessary functions and operations to make the job well done.

1-2 Syntax Analysis:

This stage checks the syntax of the language, to make sure it follows its rules, all kinds of errors are to be reported, possible warnings could also provide information to the user.

MiniC++ like other programming language with its context free grammar expressed in BNF

Yacc is used to generate the necessary programs to recognize the program tokens into its grammar rules.

Yacc works with lex to get its tokens, also some additional information and functionality is provided by both like line-column number tracking and simple syntax error recovery.

1-3 Semantic Analysis:

This is the stage where each token is giving a meaning with conjunction with other tokens, we check the constructs to be rightly composed, type checking is the essence of this stage, and that is:

- does this function exists
- does this function call matches any previous definition or declaration
- is this variable a class member
- are the operands of this operator acceptable
- assessors, scopes, classes and inheritance..
- .
- .

And much more.

Whatever the error in this stage, type checking must continue till the end of the program to discover as much semantic errors as possible.

It's also the stage where these tasks are accomplished or completed:

- Symbol Table construction and management, inside this table all variables and frequently used information, functions, classes are stored with their associated information.
- Building or completing the Abstract Syntax Tree which represents the whole program and helps type checking and later stages in code optimization and generations.
- And finally Type Checking, which represents the most difficult and time consuming task in our project.

1-4 Code Optimization

In the optimization phase, the compiler performs various transformations in order to improve the generated code. These transformations will result in faster-running machine code.

Code optimization is not restricted before or after code generation phase, few optimization processes could be done over the AST to improve the tree and cut some nodes off calculations, the remainder of this process could be mapped as finding a way to make a piece of code to run faster, like loop optimization, dangling expression exposure, compile time constants calculations, the consecutive ANDs, ORs cut.

1-5 Code Generation

The final phase in the compilation process is the generation of target code. This process involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task

Our generated code will output a VM code of a stack runtime environment, consists of simple statements equivalent to the input code.

For each node in the AST the equivalent code is generated to get a complete running program.

2- Lexical Analysis:

Basically We'll describe the Lex file content plus the main operations to get our tokens right.

- keywords
- Operators
- Include file processing
- Class Type – Variable Type discrimination
- Comments handling
- Line-column tracking

Spaces, new lines and tabs are ignored.

keywords		Special characters	
class	while	(^
cin	delete)	
cout	friend	,	?
break	inline	#	:
char	new	{	;
const	operator	}	=
define	protected	[::
do	private]	->
double	public	.	++
else	this	&	--
extern	false	*	<=
for	true	+	>=
if	bool	-	==
int	NULL	~	!=
return	typedef	!	+=
continue	undef	/	-=
static	void	<	*=
		>	/=
		<<	&&
		>>	

```
identifier [a-zA-Z_][0-9a-zA-Z_]*
```

```
exponent_part [eE][+]?[0-9]+
```

```
fractional_const ([0-9]*"."[0-9]+)|([0-9]+".")
```

```
double_const (({fractional_const}{exponent_part}?)|([0-9]+{exponent_part}))
```

All lexical rules are straight forward and no need to describe them, however some took some effort and work which we'll describe.

2-1 Include File Handling:

flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<x>" will only be active when the scanner is in the start condition named "x". two methods for start conditions: inclusive and exclusive.

In exclusive, only rules qualified with the start condition will be active and it is what we used for the include, and comments handling.

The exclusive start condition was used just to capture the include name properly, and these are the rules to capture this name:

```
#include                                BEGIN(INCLUDE_STATE);
<INCLUDE_STATE>"<"                     BEGIN(INCL_S0);
<INCLUDE_STATE>[ \t]*                   { }
<INCLUDE_STATE>[^<\t \n]*"\n"           BEGIN(INITIAL);
<INCL_S0>[^>\n]+                         {
    strncpy(fileNameBuffer,yytext,yylen);
    fileNameBuffer[yylen]='\0';
    BEGIN(INCL_S00);
}

<INCL_S00>">"                          {
    if(!includeFileProcess()){cout<<"include file
doesn't exist\n";}BEGIN(INITIAL);}
<INCL_S00>"\n"                          {
    BEGIN(INITIAL);
    yyerror("include file is not well declared,
it's ignored\n");
}

<INCLUDE_STATE>[" ]+                     { BEGIN(INCL_S1);}
<INCL_S1>[^"\n]+                         {
    strncpy(fileNameBuffer,yytext,yylen);
    fileNameBuffer[yylen]='\0';
    BEGIN(INCL_S11);
}

<INCL_S11>[" ]+                          {
    if(!includeFileProcess()){cout<<"include file doesn't
exist\n";}BEGIN(INITIAL);}
<INCL_S11>"\n"                          {BEGIN(INITIAL);
    yyerror("include file is not well declared,
it's ignored\n");
}

<<EOF>> {
    if ( --include_stack_ptr < 0 )
    {
        yyterminate();
    }

    else
    {
        yy_delete_buffer( YY_CURRENT_BUFFER );
        yy_switch_to_buffer(
            include_stack[include_stack_ptr] );
    }
}
```

But First we define the states as exclusive ones

```
%x INCLUDE_STATE
%x INCL_S0
%x INCL_S00
%x INCL_S1
%x INCL_S11
```

It sounds a little bit freaky, but we implemented it and worked just fine, the previous code includes no location tracking to make it as minimum as possible, and this is the explanation:

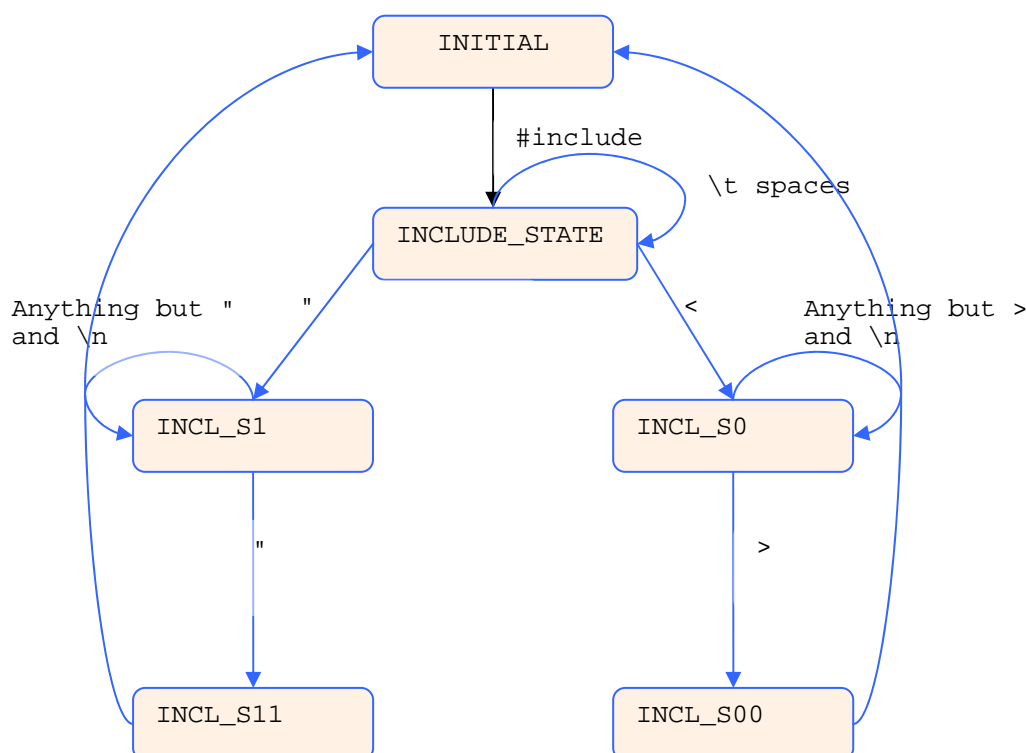
BEGIN(INITIAL) allows the rest of lex states to be considered, how ever as long as we are in one of the previous states no other statement is considered.

We began by

```
#include BEGIN( INCLUDE_STATE );
```

Which tells that we enter the **INCLUDE STATE** , a state without mentioning a start condition is considered in the INITIAL start condition.

Next figure describes this process for the include file state description of the above code.



This automata gets the lexemes that forms an include file statement like this:

```
#include "hello.h"
#include <hui.h>
```

Spaces could exist between include and " or <, the enclosing must match the opening character, and that means: " matches " and < matches >.

if we reached the end of the line without having a valid file name enclosed inside such enclosing characters then we ignore the include statement and continue lexing, after notifying the user with such error, (this is one error recovery rule during lexical analysis)

After reaching INCL_S11 or INCL_S00 we try to process the extracted included file by **includeFileProcess** method which does this:

- try to open the include file for reading
- if current include depth > MAX_INCLUDE_DEPTH then stop
- switch in Lex input buffer:
yy_switch_to_buffer(yy_new_buffer(yyin, YY_BUF_SIZE))
- increase include depth

when we reach the end of the file we return to the previous file in the stack

this buffer switching mechanism is allowed by Flex and described thoroughly in it's documentation.

As seen here, the include file is handled during the lexical analysis phase, by switching input buffers nothing more, so parsing sees nothing of this operation.
This operation (handling include files) took up to 4 hours of search, try-error and work.

2-2 Class Type – Variable Type discrimination

This is the case:

The identifier lex definition matches that of a class Type, for example when we have this code:

```
class Stack { ... }
int Stack=0;
```

both have the same definition and extracted in the same rule, how ever, when we worked with Yacc and generated our rules for the class type we got a conflict problem which we could only resolved by discrimination of variables and class types during the lexical analysis and this is how we did it:

```
%x CLASS_NAME
.
.
class          { BEGIN(CLASS_NAME); return CLASS; }
<CLASS_NAME>{identifier} { BEGIN(INITIAL); addClassID(yytext); return
                        CLASS_ID; }

{identifier}   { if(isClassType(yytext)) return CLASS_ID; else return ID_NAME; }
```

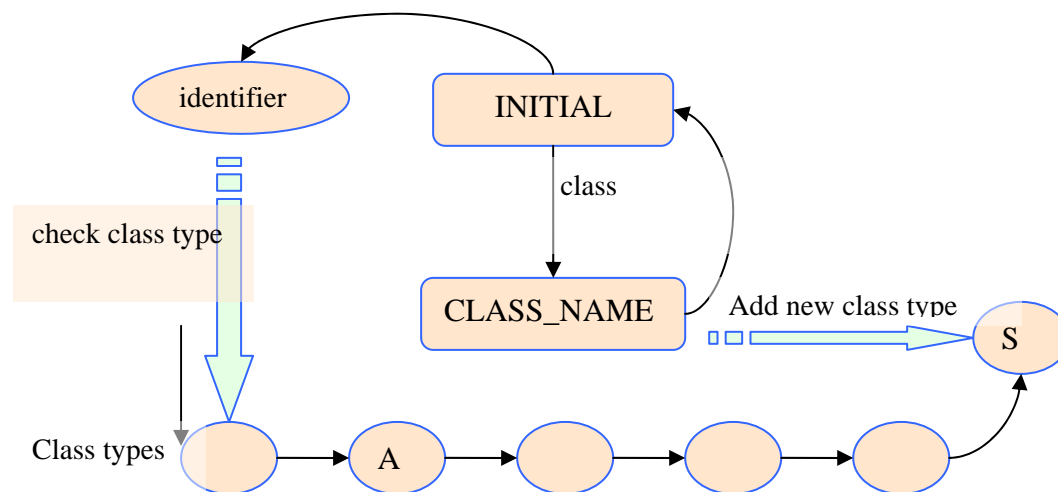
we defined An exclusive state named CLASS_NAME which could only be entered when we face a **class** keyword, so when we are in such a state we add this class Type (Stack) into a lined list of such names.

So when we face an identifier in the normal case like : Stack s=new Stack() we search this list for a match and if we find one we return a CLASS_ID else ID_NAME.

This list at the end will contain all class types in the source code.

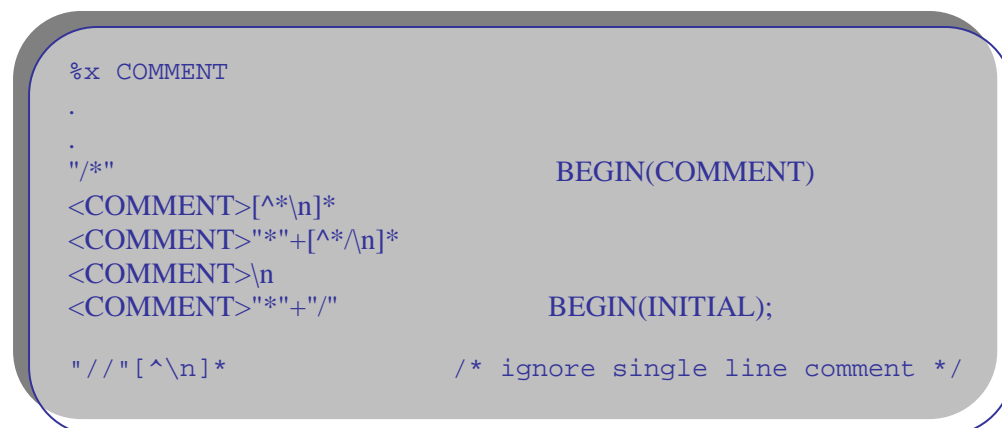
This method helped in resolving a major conflict in the grammar we written.

Next is a view of this process:



2-3 Comments handling:

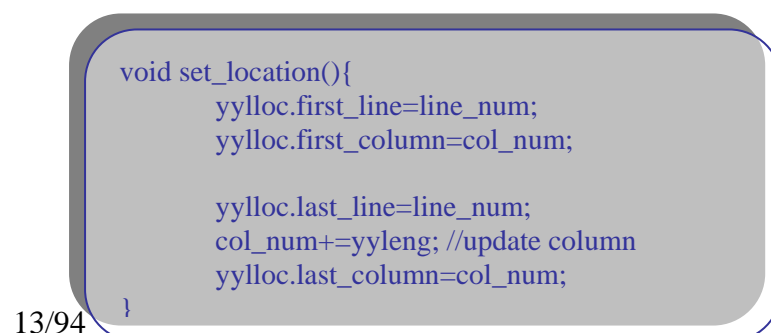
Just like we did for include and class type we do for the comments:



First we define a COMMENT start condition as exclusive one which takes control when facing /* and never stops until it drains all character and reaches an enclosing */
 So what ever the characters encountered they are skipped until the comment ends and this works fine for multiple lines, and any where comments.
 single line comments are also handled, they work normally without start conditions and eat up characters until new line is encountered.

2-4 Line-column tracking:

Upon each state change, `set_location();` method is called



Which uses Lex location tracking record which passed to Yacc, this location includes line and column number of the beginning and ending word.

each time we see a new line we increase the line_num even inside comments.

Column number is increases by the length of the current yytext which is yyleng value.

Basically this is the most important feature of our lexer, next we talk about the Syntax Analysis phase and the grammar.

3- Syntax Analysis:

The grammar was written in Yacc and the Yacc implementation was under Bison, so there was some features of Bison we took advantage of.

3-1 Conflict resolving approach

we wrote the grammar completely from scratch and it took up to 3 days to eliminate conflicts at first trial we got 170 shift-reduce conflicts and 23 reduce-reduce conflict.

many of them was simply: one leads to another at the end of conflict resolving it yielded no conflicts at all.

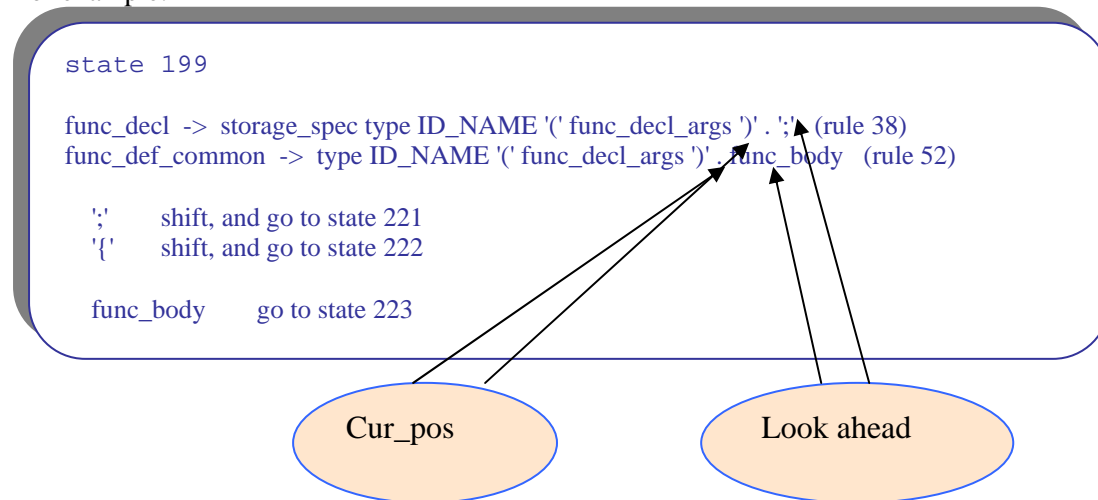
To get conflict information we command bison to run in debug and verbose commands, you can define:

```
#define YYDEBUG 1
```

```
#define YYERROR_VERBOSE 1
```

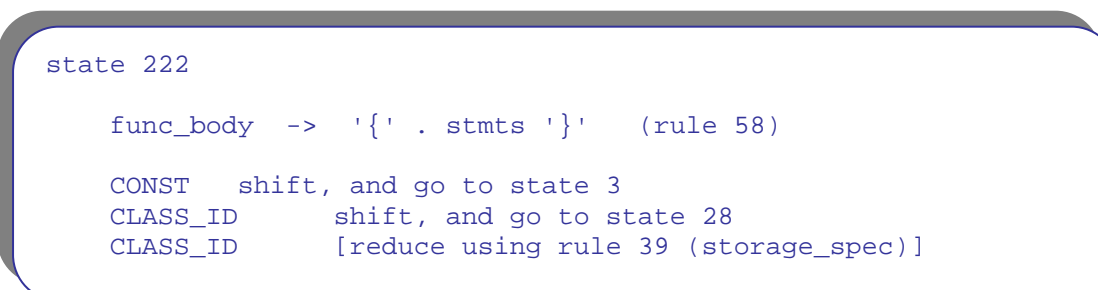
Or when running bison we run it with `-v` flag (verbose) to output such a file, which contains a description of all states of the grammar, current position inside a state and the look ahead character.

For example:



here for example: we are at position . (before ; and body) the dot refers to current location in the state, so we got one of two choices to shift and go to state 221 when look ahead is ';' or to state 222 when '{' is the look ahead

how ever if using the same look ahead we got two cases (shift or reduce) then it's a conflict.



Here `CLASS_ID` is a look ahead and we got Shift-reduce conflict which must be resolved. `CLASS_ID` is a look ahead in one of **stmts** rules, and in `storage_spec` rule which both could be moved into according to current LA.

Next we'll mention some of the rules that forms our grammar

3-2 Sample rules:

```
program :  
    prog_decls  
    ;  
  
prog_decls :  
    | prog_decls declaration  
    | prog_decls error  
    ;  
  
declaration:  
    var_decl ';' |  
    func_decl  
    | func_def  
    | class_decl  
    | class_func_def  
    | class_def  
    | class_static_init_var ';' |  
    | class_destructor  
    | class_constructor  
    ;
```

So the program consists of one or more declarations, a declaration could be an error declaration.

A declaration could be one of:

A variable declaration: we allow declaring global variables

Also a function declaration and definition: the definition refers to implementation

Also a class declaration like: class A;

Declaring a class to tell it's later defined.

Declaration also includes class function definition, constructor and a destructor, which we differentiate between each other.

A class definition consists of:

- a class type
- possible inheritance base list and base list access specification
- and a class body

A class body consists of:

- access specification like Public, Private, Protected, Friend
- normal variable declaration
- normal function declaration
- a constructor declaration
- destructor declaration

All these rules were hand written from scratch, they identify the MiniC++ grammar wanted, but in general they are straight forward, which we didn't care whether it's a good grammar or not as much as we cared that it works or not.


```
class_def:
    class_head class_body
|class_head ':' access_spec class_base_list
class_body
;

class_head:
    CLASS class_type
;

access_spec:
    PUBLIC
|PROTECTED
|PRIVATE
|FRIEND
;

class_base_list:
    class_type
|class_base_list ',' class_type
|class_base_list ',' error
;

class_body:
    '{' '}' ';'
|{' class_body_stmts '}' ';'
;

class_body_stmts:
    class_body_stmt
|class_body_stmts class_body_stmt
;

class_body_stmt:
    access_spec ':'
|func_decl
|func_def
|var_decl ';'
|class_constructor_decl
|class_destructor_decl
|error ';'
|error '}'
;

class_constructor_decl:
    class_type '(' func_decl_args ')' ';'
;

class_destructor_decl:
    '~' class_type '(' ' ')' ';'
;
```

A class constructor could be defined like this:

A::A(params): a(1),(2){ stmts }

It includes the constructor initialization list

Destructors have no parameters and identified by the ~ letter.

```
class_constructor:
    class_type SCOPE_DOTS class_type '(' func_decl_args ')'
                                class_constructor_init_list block_stmt    ;

class_constructor_init_list:
    ':' c_c_init_list
    ;

c_c_init_list:
    ID_NAME '(' expr ')'
    | c_c_init_list ',' ID_NAME '(' expr ')'
    ;

class_destructor:
    class_type SCOPE_DOTS '~' class_type '(' ')' block_stmt
    ;

class_decl:
    class_head ';'
    ;
```

```
array_dims:
    array_dim
    | array_dims array_dim
    ;

array_dim:
    '[' ']'
    | '[' INTEGER_CONST ']'
    ;

array_init_list:
    '{' array_init_list_consts '}'
    | '{' array_init_list_lists '}'
    ;

array_init_list_consts:
    expr
    | array_init_list_consts ',' expr
    ;

array_init_list_lists:
    array_init_list
    | array_init_list_lists ',' array_init_list
    ;
```

the above declarations enables a complex multi-level array initialization, i.e.

```
int x[3][2]={ {0,1},{1,2},{0,0}};
and more.
```

We'll mention no more rules, since most of them will reveal it self when type checking is mentioned.

3-3 Shift/Reduce Conflicts and resolving:

We'll mention some of the conflicts we had and resolved, though many conflicts as we said before were kind of one leads to another.

(1)

```
if_stmt:
    IF '(' expr ')' stmt
  | IF '(' expr ')' stmt ELSE stmt
```

The conflict comes from the fact whether the parser should shift or reduce when seeing else in the following case:

```
If ( expr)
    If ( expr)
        Stmt
    Else
        Stmt
```

So in this case, when reaching **Else** should we shift to match the previous if or should we reduce the previous if into a statement and for else to be matched with the first if?

This simple conflict could be easily be resolved by defining a precedence for the second if-else higher than the first if.

But even more simpler: is to do nothing at all.

Bison, by default shifts, so this Shift-Reduce conflict is automatically resolved by the parser and no need to define precedence or anything, this happens because of the look ahead char, which helps it to make the right choice in such case.

(2) Discrimination between function declaration and definition arguments caused reduce/reduce conflicts which resolved them by unifying the two rules.

(3) declaring class variables can't be done with empty arguments, it conflicts with the Proc_empty_args and thus makes epsilon reduction so the form A a(); is incorrect and must be replaced by A a; where A is class type and has a default constructor with empty params the declaration: A a(1,2); how ever is allowed when constructor has params we can enable the A a(); form but then we have to separate class type from other types and make separate rules which we wont do.

(4) int A::x=0; in static member initialization, it's
class_static_init_var:

type class_type SCOPE_DOTS ID_NAME '=' expr;
 int A::x=0; in static member initialization, it's a unique variable initialization rule
 only happens in a global scope
 putting the rule in the following form used causes Shift-reduce conflicts:
 type class_type SCOPE_DOTS ID_NAME '=' expr
 which later was resolved after few changes in the class_type

(5) class type: each defined class is considered a type it self, this is why any Identifier is considered as a possible type so discrimination between class types and identifiers was done in lex.

The most common conflict we got was the Epsilon reduction conflict, which we had to stretch some rules for it to eliminate it.

Many more conflicts were found and resolved, but we can't remember them all, nor we did documented them at the time of being due hurry, besides, some times fixing one rule resolved over 80 chaining conflicts.

3-4 Error recovery :

In the lexical stage: simple error recovery by:

- unrecognized characters are skipped
- unwell formed include statements are ignored

in the grammar:

we tried to make error recovery as light as possible, since jamming the grammar with error recovery rules, could it self double the size of the LR table and double the states. We'll mention some of these rules:

```
prog_decls :
|         prog_decls declaration
|         prog_decls error
;
```

A main program declaration error could skip a major program error.

Error recovery rules could be added any almost any where as long as they cause no conflicts .

Examples of error recovery rules:

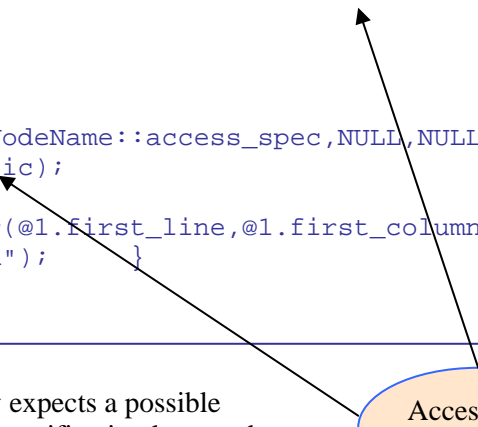
```
class_def:
  class_head class_body
|class_head ':' access_spec class_base_list class_body
  {$$_new TreeRecord(@1,NodeName::class_def,$1,$3,$4,$5);}
|class_head access_spec class_base_list class_body
  {$$_new TreeRecord(@1,NodeName::class_def,$1,$2,$3,$4);}
  ErrorReport::printError(@1.first_line,@1.first_column,"missing :\n");
;
```

This is the simplest kind of syntax errors recovery rules: try to expect the single letters that could be missed or altered and create an error recovery rule for it like:

Missing ':'
 Missing ';'.

Substituting ';' with ',' or vice versa
 Missing (
 Replacing { with [or (and
 much more...

```
access_spec:
    PUBLIC
    {$$=new TreeRecord(@1,NodeName::access_spec,NULL,NULL,NULL,NULL,
                        Type::access_public);}
    | PROTECTED
    | PRIVATE
    | FRIEND
    | error
    {$$=new TreeRecord(@1,NodeName::access_spec,NULL,NULL,NULL,NULL,
                        Type::access_public);}
    ErrorReport::printError(@1.first_line,@1.first_column,"unknown access
    specification :\n");
    ;
```



This type of error recovery expects a possible Error in writing an access specification keyword For example to write private without 'e' or to write It with capital letter, here any error in writing as access_spec keyword is substituted by a default access_spec rule and that is the PUBLIC

```
class_base_list:
    class_type
    | class_base_list ',' class_type
    | class_base_list ';' error
    | class_base_list ':' error
    ;
```

An error in the class base list like:

Class A: public B, C{ ... }

Where C is not a predefined class type

Or putting ';' instead of 'm'

Also the if for or even while statements which we could miss a '(' or anything :

```
if_stmt:
    IF '(' expr ')' stmt
    | IF '(' expr ')' stmt ELSE stmt
    | IF expr ')' stmt {...
    ErrorReport::printError(@1.last_line,@1.last_column,"missing ( :\n");};

for_stmt:
    FOR '(' var_decl ';' expr ';' for_itr ')' block_stmt
    FOR var_decl ';' expr ';' for_itr ')' block_stmt {...
    ErrorReport::printError(@2.first_line,@2.first_column,"missing ( :\n");};
    ;
```

```
stmt:
    simple_stmt ';'
    | compound_stmt
    | error ';'
    | error '}'
    ;
```

This rule recovers from a serious statement error, we expect that when error happens inside a statement state a ';' or '}' will end this statement and recover the rule.

Of course not all syntax errors could be recovered from for an implementation costs and conflict reasons, and even the commercial compilers suffers this, include Borland Compilers and MS VC compiler, for example:

VC compiler doesn't recover from ';' when it's missing at the end of a class definition, and enters an error loop and the stack overflows to tell you that your program exceeded max errors allowed, even when your program contains only one or two errors besides this missing semicolon.

3-5 Location Tracking :

As we said before in the Lexical Analysis section, location tracking is managed there and passed in a standard yylloc struct which contains 4 fields:

```
void set_location(){
    yylloc.first_line=line_num;
    yylloc.first_column=col_num;

    yylloc.last_line=line_num;
    col_num+=yyleng; //update column
    yylloc.last_column=col_num;
}
```

This struct is passed to the Yacc and accessed by using @location (@1.first_line)

This is too passed to the AST to include location information too.

- Association rules are defined by using %left macro
- type of nodes are defined by using %type macro
- .
- .

One final note about the Yacc grammar rules:

We build our Abstract Syntax Tree during grammar parsing, how ever this tree only contains the nodes of the tree, without any Symbol table connections, and no other processing is made during parsing.

4- Semantic Analysis:

The time consuming and most difficult phase in our project.

It includes:

- Symbol table creation and management
- AST management
- Connecting the AST with the Symbol Table
- Type checking of all program constructs

4-1 Symbol Table management and creation:

We needed a structure that enables us to store and retrieve certain information efficiently, such information which is to be used frequently or needs to have one instance in the whole program, the symbol tables records are to be referenced by the AST nodes.

It's also responsible for scope management information.

This information is used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements. The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.

4-1-1 What information do we store inside this symbol table?

- Variables and their information
- Functions, it's return type, it's signature information and a reference to the AST node that this function is.
- Class names and their reference nodes
- Scope management information
- Code generation related information
- Array info
- Access specification info (like public...)

4-1-2 Symbol Table structure:

Our choice of the symbol table was a Hash Table.

One could create a symbol table for each program block, for example:

Every time a function is encountered a new symbol table is created, symbol tables are linked with each other to assemble scope information in the program.

How ever we did it in a different way:

For the whole program we created one hash table, one symbol table, and used additional information to store scope information, the following diagrams explains how this was done for the code presented bellow:

Variables and functions defined inside a class are considered inside it's scope

The file has it's global scope

Each function has it's scope

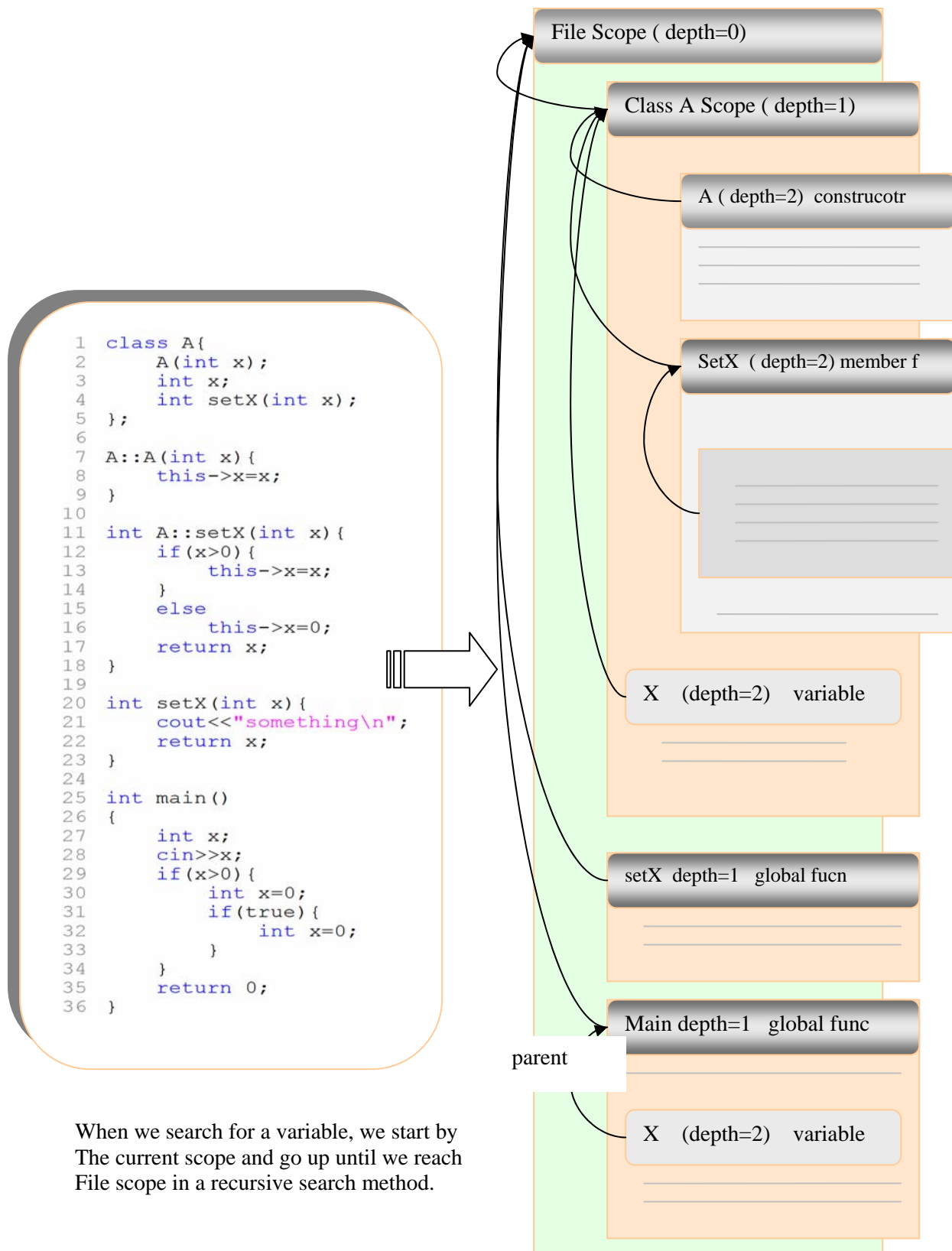
Also each block has it's scope

What unifies a scope info is: it's depth and parent:

For example

In the code bellow, two variables named x with depth 2 ,one the parent record was main and the other was the class A.

Two function named setX: one depth 1 and the other depth 2 and they both have different parents.



When we search for a variable, we start by The current scope and go up until we reach File scope in a recursive search method.

Each time we enter a new scope, we increase Depth value, and we decrease it back when we exit the scope.

We assume that functions and classes are always defined in the global scope (file depth + 1)
This scope management method works for any depth and nested names and unnamed blocks.

This is an example of the symbol table content of the previous script, we print the hex address of a symbol table record, also the parent to show (if you were ready to stare) this scope info:

*****symbol table content*****

rec_address	parent_address	depth	type	name
0x00502790	0x00502450	4	111	x
0x00502158	0x00501E18	3	111	x
0x00501E18	0x00501748	2	118	1_block_
0x00501A88	0x00501748	2	111	x
0x005014F8	0x005011B8	2	111	x
0x005011B8	0x00321338	1	113	setX
0x00500B50	0x005003B0	3	111	x
0x00500690	0x0032FEA0	3	111	x
0x005003B0		2	113	setX
0x00500160		2	111	x
0x00321338	0x00000000	0	0	__File_Scope_Rec__
0x00502450	0x00501E18	3	118	2_block_
0x00501748	0x00321338	1	113	main
0x00500E90	0x005003B0	3	118	0_block_

*****prog tree content*****

Record type(var,
func, class...)

Record name: like the
id name or class name

4-1-3 Symbol table Content Sample

Above for example:

Three records: member function, member variable and a constructor which have one parent record : a class record.

Also:

7 records for the variable x of type 111 (int), each one has different scope, even the ones that are a function parameters are considered as children of the function in a scope sense.

4-1-4 The hash function :

We made the hash function as simple as possible, which depended on the string name sum of multiplication with depth value, so in general, no two similar names will have the same value. As for comparison of two records when adding or finding: we depend on the <name, depth, type, parent>

Since we could have a function and a variable with the same name, the same depth, the same parent, but they'll differ in type.

That's all about Symbol table, some additional information about it will be mentioned in the next sections.

4-2 The Abstract Syntax Tree:

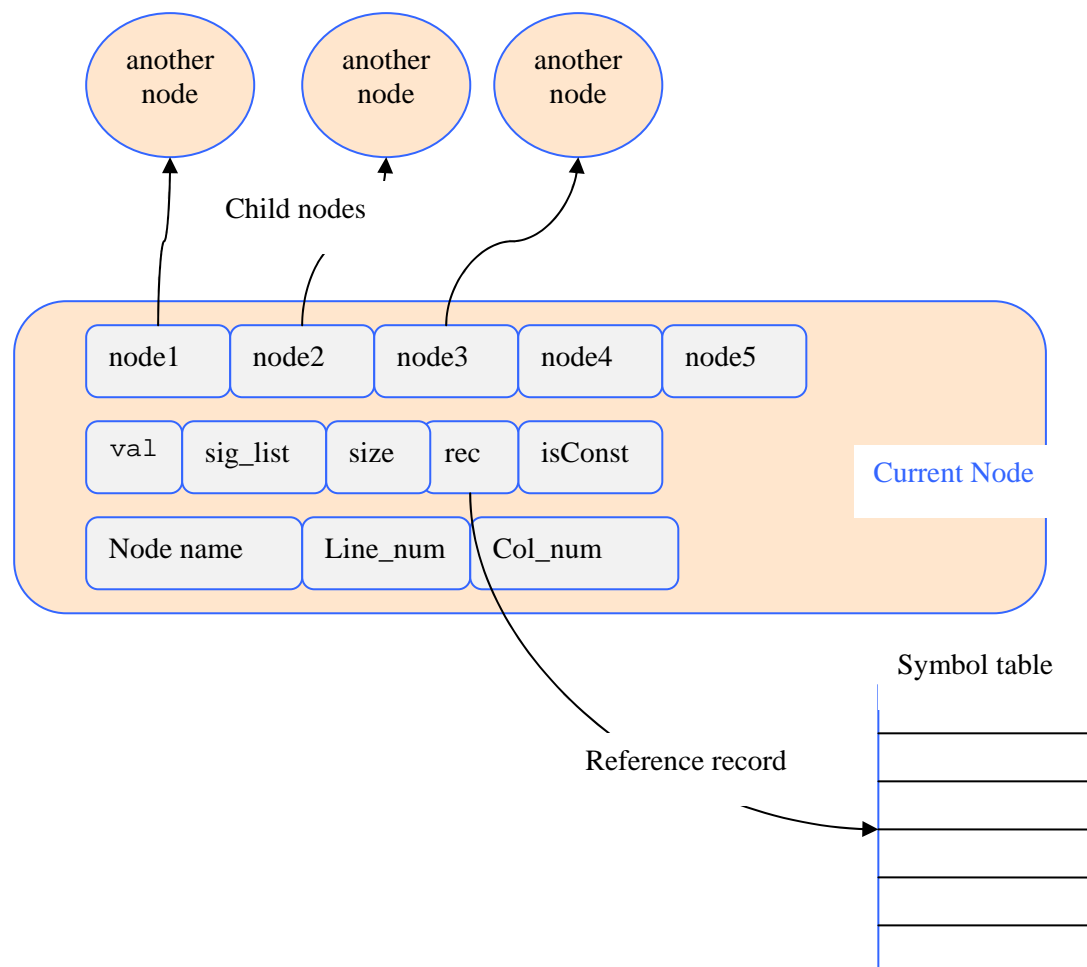
The AST will contain the program code, formatted in a way to make it easy to reach, traverse and check the nodes and types.

What ever the error in the program, we must continue checking until the whole tree is checked.

We choose a generalized tree, not a binary one, each node chooses what fields to fill and what to leave depending on it's need.

4-2-1 Node Structure:

A node could be described like this:



nodei: are child nodes.

sal: is a union to hold constant data values

sig_list: is a linked list to hold the type of current node (explained later)

size: (code generation specification)

rec: is the reference to the symbol table if it's a leaf node, any node that has data in the symbol table like functions, classes...

isConst: used for optimization to tell if current expression is const or not, when it's composed of more than one const children.

Next, we'll describe each node, along with it we'll describe the type checking made on such node.

4-3 Type Checking & AST Nodes:

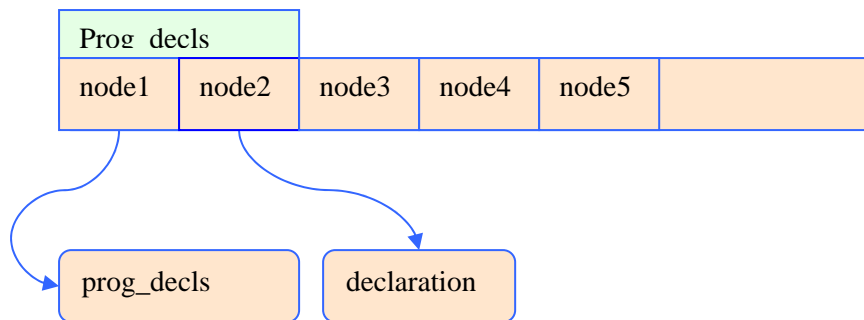
We transverse the AST and check every node before it's considered ok.

these checks are mapped to the grammar declaration syntax, the traversal adds the symbol table info, and updates it.

All next mentioned nodes and type checking notes are fully implemented and tested.

A program consists of a prog_decls

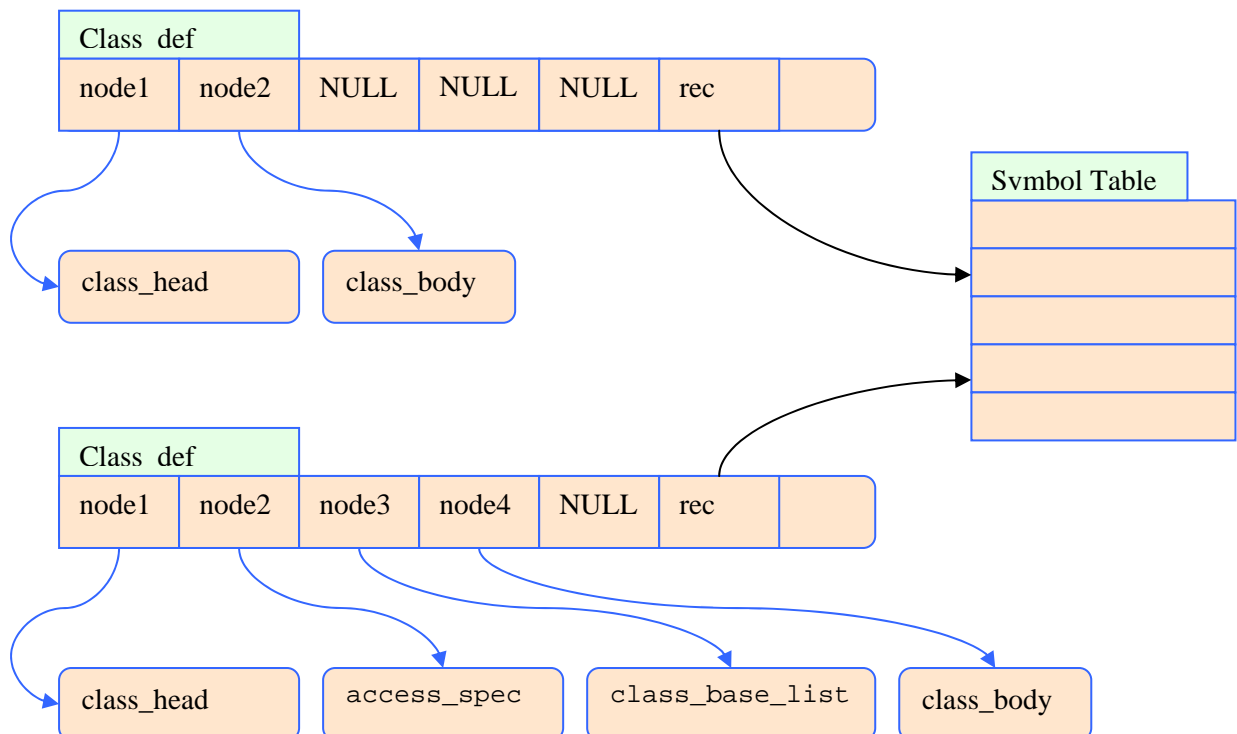
4-3-1 first node is the prog_decls which is one or more program declaration



Here check is forwarded in the first two children.

Declaration has many forms, include global variable declaration, class definition and declaration and much more, each one has it's own node structure and it's own check model.

4-3-2 class definition node



as seen here is a class definition node, which depending on NULL of the third node we know which part it has.

A class definition could have this form:

Class A{ ... }

Class A: public B,C,D{ ... }

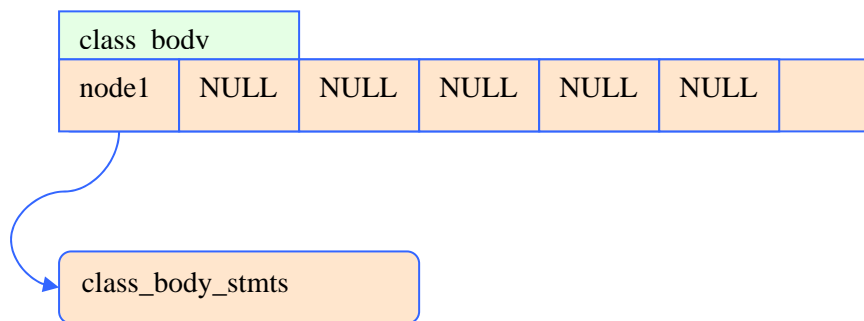
This is what we do when facing this node:

- check the head by adding the class type to the symbol table, if it's already there due previous declaration (only) then we update the node_ref pointer of the relevant symbol table record
(i.e. the case where it was declared as class A;)
- if a class has been defined with the same name then we issue an error message
- check class_base_list node not to contain names of the same derived name, since You can't derive from you self, also make sure that no recursive inheritance exists, in Yacc it was checked that base list are actually classes, so no need for this check again.
- forward check to the class_body node.

4-3-3 class body node:

When seeing a class body rule, we create new scope, which is the be the class scope, and the current parent becomes this creates scope (class record)

When we exit this class body { } we restore old parent and old depth by decreasing depth and setting the current parent to the parent of the parent.

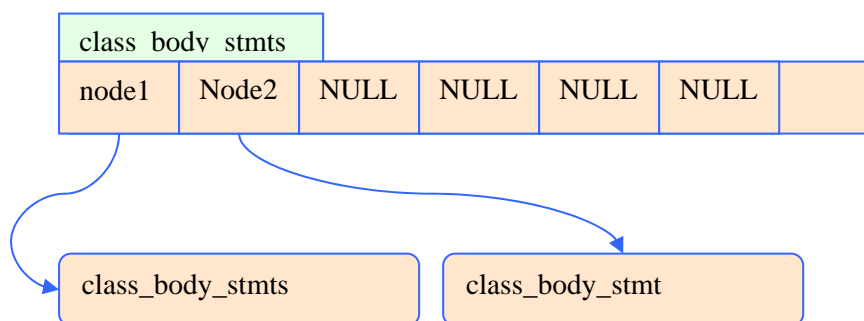


Also when facing this node the current access specification is set to Private by default for all consecutive nodes to come until it's explicitly changed.

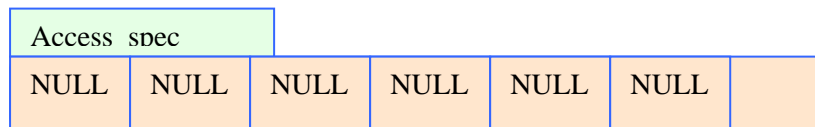
access info is stored in the symbol table for fast access.

The class body has one child: it's statements, (sure it could be eliminated, but we preserved it to preserve for old implementation reasons, which we didn't have time to change)

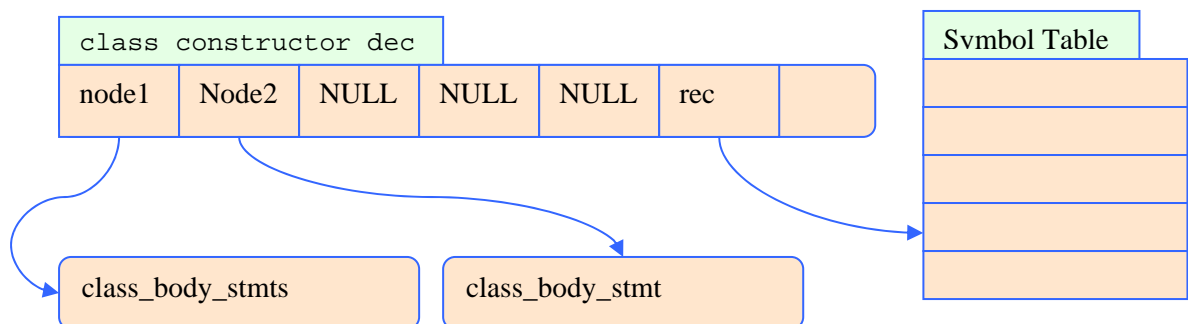
4-3-4 class body statement node:



This node consist of one or more class body statements, so we forward the check to the same check on the first child and the second child for one statement.

4-3-5 access specification node (public , protected...):

This node only contains a type info, and no children, so it's a leaf node, upon this node we change the current access_spec to this node's value, so all consecutive nodes will have this access_spec value

4-3-6 class constructor declaration , Function Overloading, Function Signature coding:

This is declaration only, inside a class definition like:

Class A{ A(int x); ... };

- first of all check if this constructor is the current class constructor by matching it's name to the current parent class.
- check if it was already declared before with the same signature by searching the symbol table for records of the same name, depth, parent, and signature
- add the declaration to the symbol table, set it's scope and access spec, code it's signature, and set the parent to the current parent.

So multiple declarations of constructor are allowed but with different signature (function overloading)

Function Signature coding & Function Overloading:

So we'll explain how function signature is matched against another function signature:

- function signature depends only on it's parameter declarations
- f(int,int) conflicts with f(int x,int y) or f(int s,int v) so we use this signature to enable function overloading
- function call uses the same method to match it's call signature against it's declaration Signature
- signature info is integer list for fast matching signatures

In our project we used a fine and fast method to match types against each other, the same method was used to match function argument signatures.

It's by defining a type as list of integers, so for example:

int	101
double	102
pointer	107

And so on for each type, so when we got a complex type like this one:

Int **** the type signature list will become 101 107 107 107 107

As for classes: A* a the signature is [A record address] 107

By this method we can match two types against each other as fast as possible in spite their complexity

The same method is used in coding function signature, by appending each argument type to the function signature list for example:

void f(int, int, double) has the signature 101 101 102

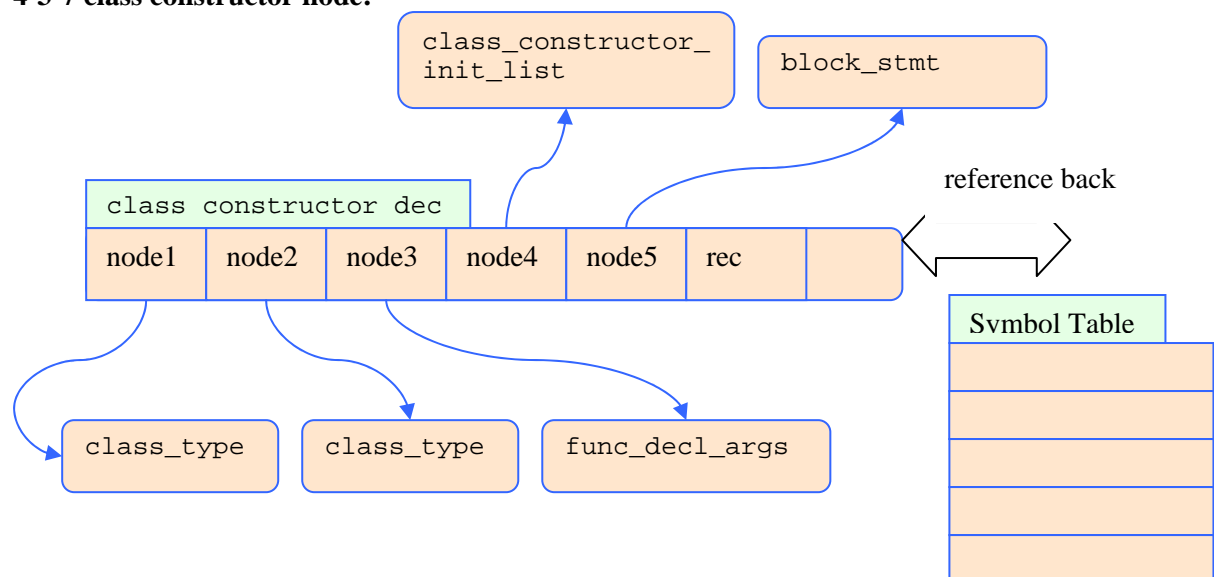
int f(int*,double*, int) has the signature 101 107 102 107 101

so when matching two functions against each other we match their names, parent, depth and their signature and this allows the function overloading and fast search and match against functions.

Int ****	101 107 107 107 107
A* a	[A record address] 107
void f(int, int, double)	101 101 102
int f(int*,double*, int)	101 107 102 107 101

We could also extract single elements back and know what types they are so it's two way method for type, but for functions it's not.

4-3-7 class constructor node:



Class constructor implementation like:

```
A::A(int x,int y):val(0){ ... }
```

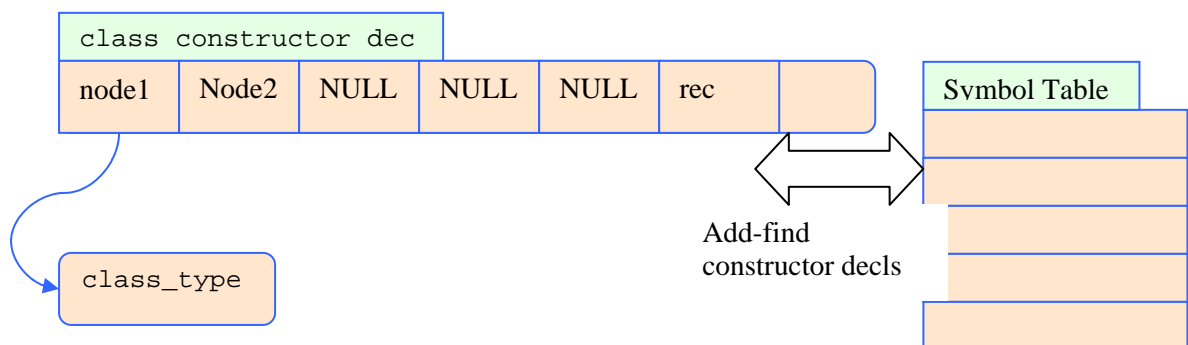
We need to make these checks:

- a constructor must match a declared constructor in a defined class(match name and signature)
- so we search the symbol table to find a record with the same name of our constructor, with the same parent as the class_type and the same signature, the

record we find must contain a reference node of NULL which indicates that the record is reserved only for declaration.

- If we found such a record but the referencing node was not NULL then this constructor is a redefinition with same signature and that is not allowed.
- Then if declaration only was found, variables inside the declarations arguments must be stored in the symbol table with the relevant function scope, and the nodes they come from must reference them back.
- We check the member initialization list to see that the members are class members or not.
- The symbol table record is updated by setting it's reference node to this constructor node.
- This constructor has scope of it's own for it's internal statements, so we update the current parent to be this node and increase the depth
- After that we move into checking this method body by forwarding the check into a block_stmt node check.
- After checking block_stmt we restore current_parent and depth by setting current parent to the file scope, since the parent of the parent is the class and we need to go back into the file main scope.

4-3-8 class destructor declaration node:

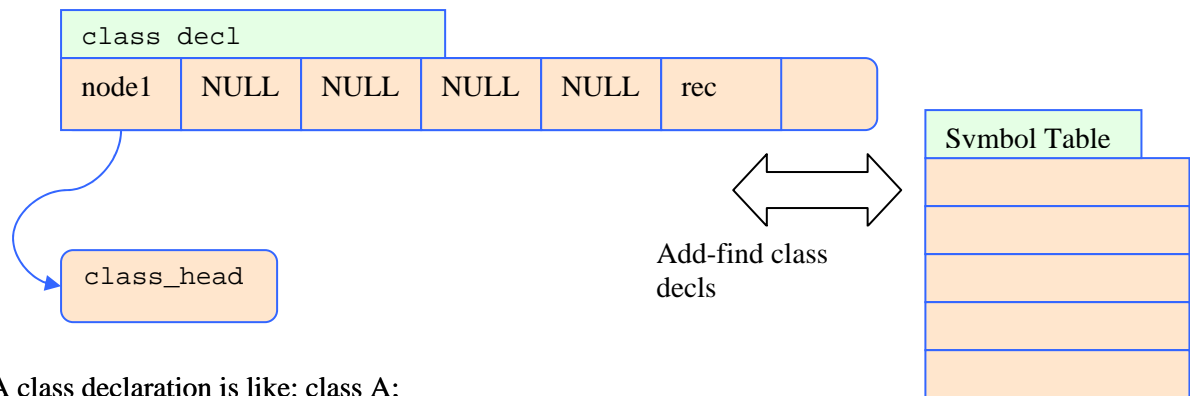


Like: `~A();`

Part of class body statements.

- when we find a destructor declaration we check that it has no parameters
- it's names matched the current class (current parent)
- no previous destructor declaration is added in the symbol table, since on e destructor is allowed.
- no reference back added in the sym tab rec (delayed until a definition is found)

4-3-9 class declaration node:



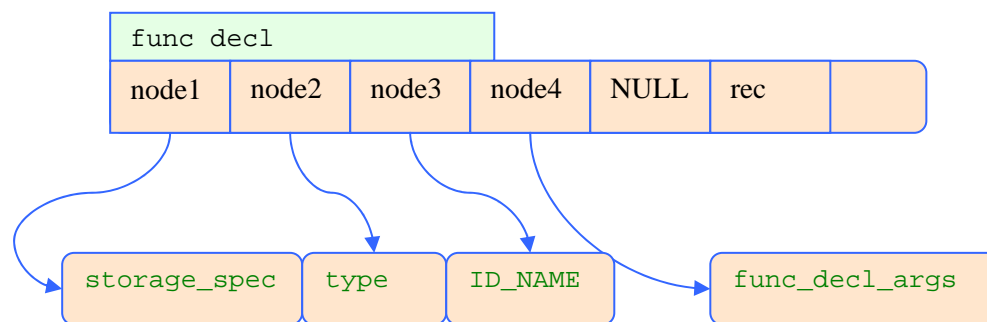
A class declaration is like: `class A;`

Just to declare it, so what we do is make a check on the `class_head` node, if a previous declaration is found then it's an error case, however if a definition was found it's fine, we do nothing, if no previous declaration or definition is found we add a new record into the symbol table with this declaration and do no referencing.

4-3-10 function declaration node:

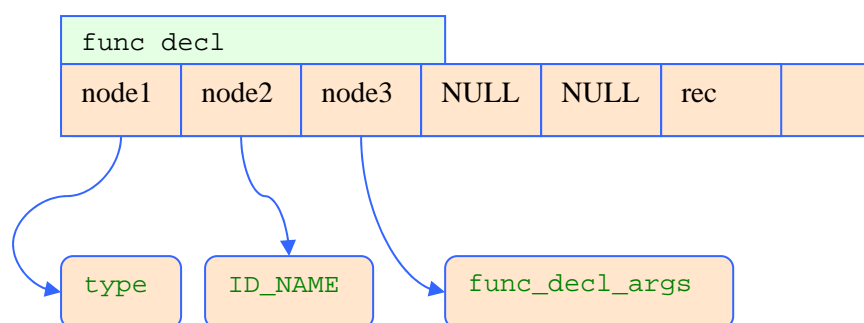
A function declaration could have these forms:

Static `int f(int x);` and the corresponding tree record would be like



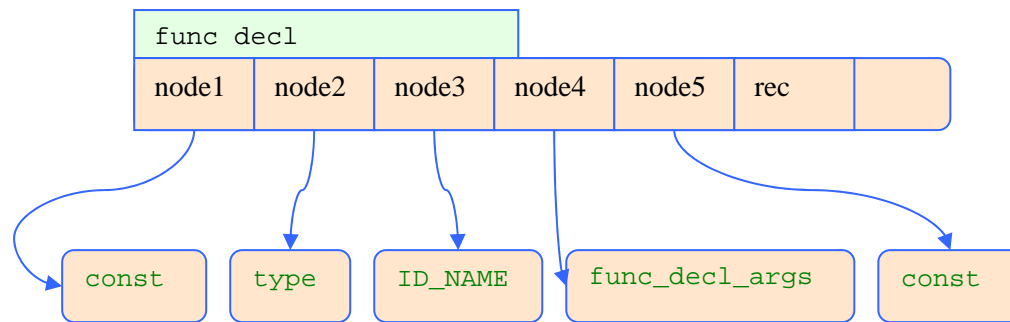
Or it has the form:

`int f(int x);` so the node becomes



Or :

`const int f(int x) const;`



Or `int f(int x) const;`

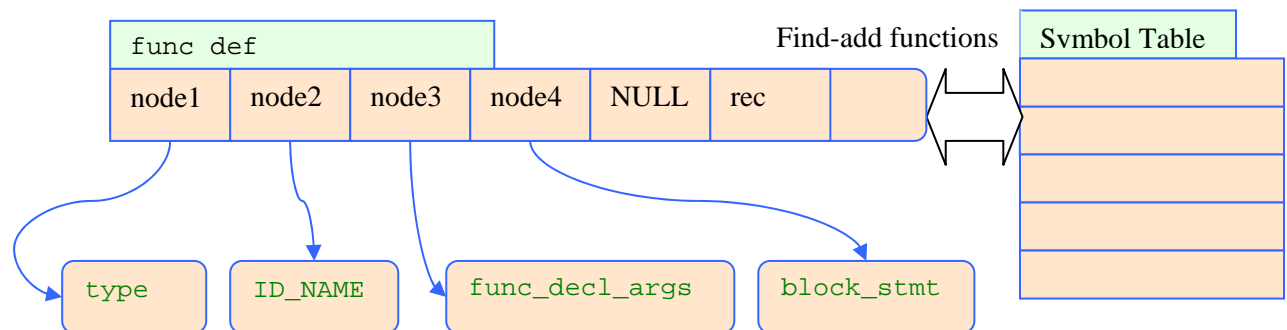
When we see a function declaration we do this:

- check if it was already declared before with the same signature
- allow multiple declarations but with different signature (function overloading)
- add the declaration to the symbol table if it's not there, set it's scope and access spec, return type.
- No special handling for `const` or `static` was made.

4-3-11 function definition node:

Function definition, could be inline or normal, and both declarations have a common part which we'll mention it alone here and without the `const` rule (too much rules to be mentioned and check against) .

i.e `int f(int x){ ... }`



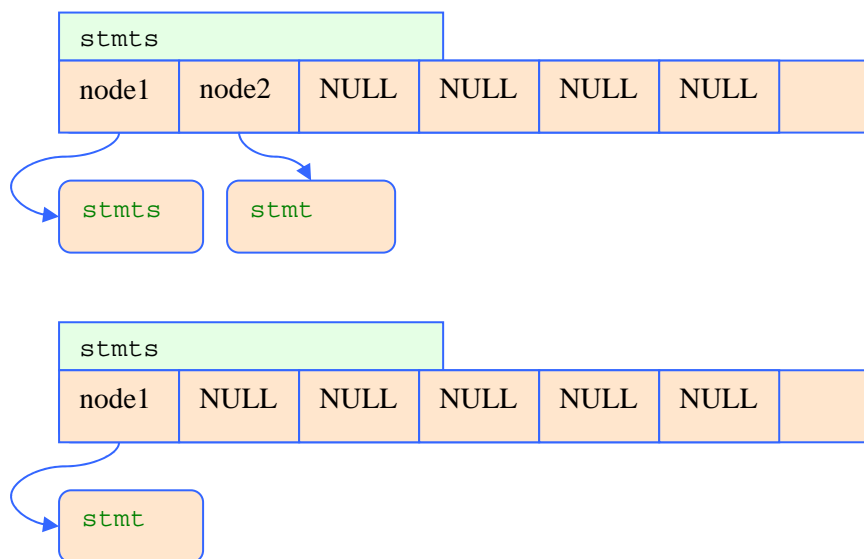
- match the definition to previous declaration if possible to update it's info matching is done using the function name and signature.
- if function is new insert the entry in the symbol table and update the reference node to reference this node back.
- add argument variables to the symbol table as part of the function scope
- set the return type of the function
- forward the check to the `block_stmt` node to check it's statements after increasing depth and setting this function record as the current parent due a new scope creation.
- When done checking the `block_stmt` restore old scope info.

4-3-12 class function definition node:

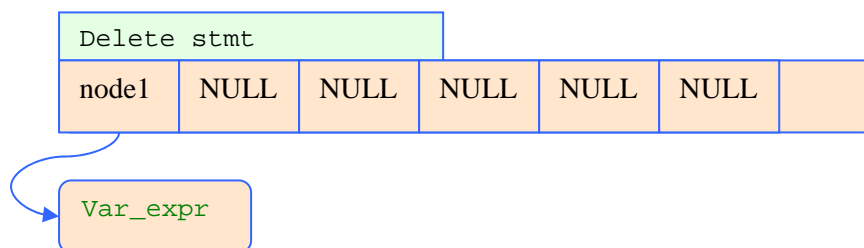
Almost the same as normal function definition but with the difference that parent of the function is the class type: `int A::f(int x){ ... }`

So declaration of the class function definition must exist, else it's considered a wrong definition.

After ending of checking this function, scope is returned back into the global file scope.

4-3-13 statements node:

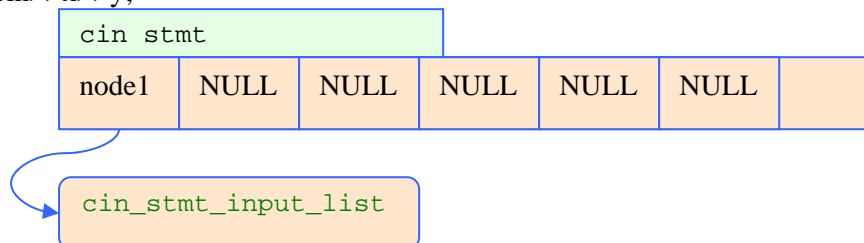
usually consists of one or more statements, this node could have only one child node of as stmt node, or two child nodes, the first stmts node and the second stmt node check is forwarded into the children to check them selves.
a statement could be a simple or compound statement, a that is to be next.

4-3-14 delete statement node:

A delete statement node has one child node and that child node must be a variable expression node, check is forward to it to check it's validation.

4-3-15 cin statement node:

Cin>>x>>y;

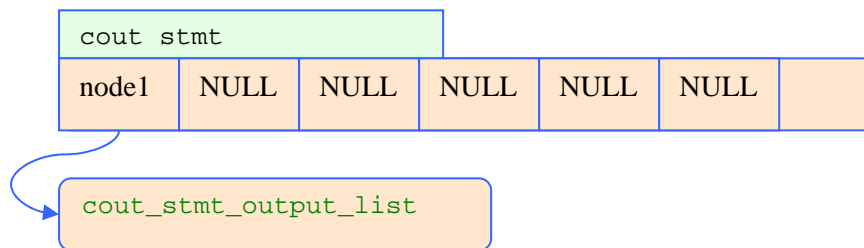


We'll not put further nodes for the cin descendants

Cin statement node has one child which is a list of one or more input elements, each element is a variable expression that could store values inside, check is forwarded down.

4-3-16 cout statement node:

```
Cout<<5<<"ddd"<<x+x;
```



The output list consists of one or more out put elements each of is an expression.

4-3-17 block statement:

```
{...}
```

Which could have 0 or more statements

4-3-18 variable declaration nodes:

```
Const int x=0;
```

```
Int x=0,y=1,z;
```

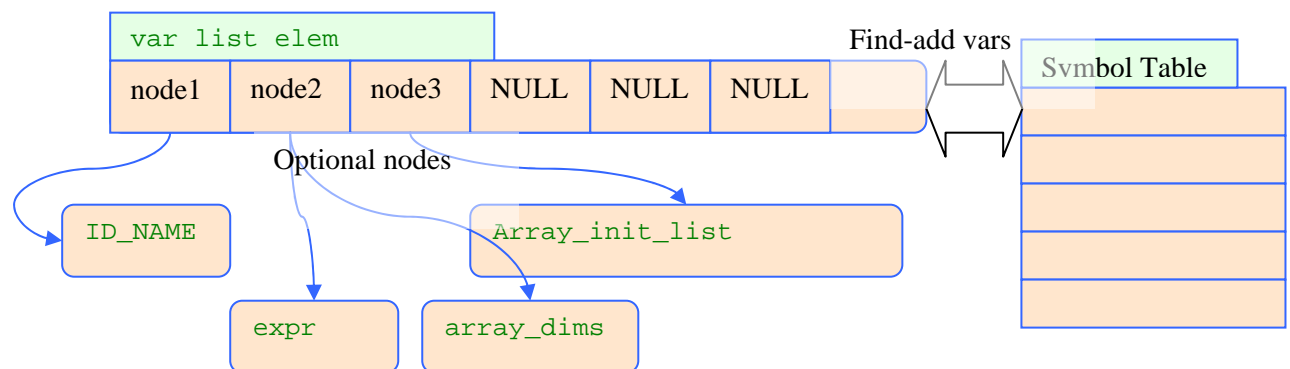
```
int x[]={ 1,2,3}
```

```
A a=new A(2);
```

- a variable declaration could have a storage specification (static, const..)

- it has a type and a variable declaration list node, which is one or more variable declaration elements.

More than one form for the declaration of a variable, here in the drawing we'll merge it into one.



A variable list element, is one element in a variable declaration list for example:

```
Int x=0,y;          y is on element.
```

- determine variable declaration type:
- if it was a normal variable declaration, then just search the symbol table for previously declared variables with the same name, if found then issue error message
- if not found then add it to the symbol table with it's scope, access spec, and type info
- if it has an assigned expression then match their types
- if it was an array declaration then make an array check
- if it was an object like A a(2); then:
 - o find the constructor that matches the type and check the procedure call parameters to match a constructor declaration in the later class.

- Classes exists in the global scope.

Now in the array variable declaration case we make a separate node for that.

4-3-19 array dims node, array information storage, and array initialization list handling:

The check against an array declaration like:

```
int a[3][2][5];
```

array information is added into the symbol table as number of single dimension, so a class was made for a single array dimension to hold such info (the dimension number and dimension size) and inside the symbol table record you find a list of this class objects to hold the complete array information so:

if we got this array declaration:

```
int a[4][5][2][3];
```

as for the type of this array in type checking calculations we assume each dimension as a pointer to the type of the next dimension.

And the variable signature list is used for type checking calculations and matching with others

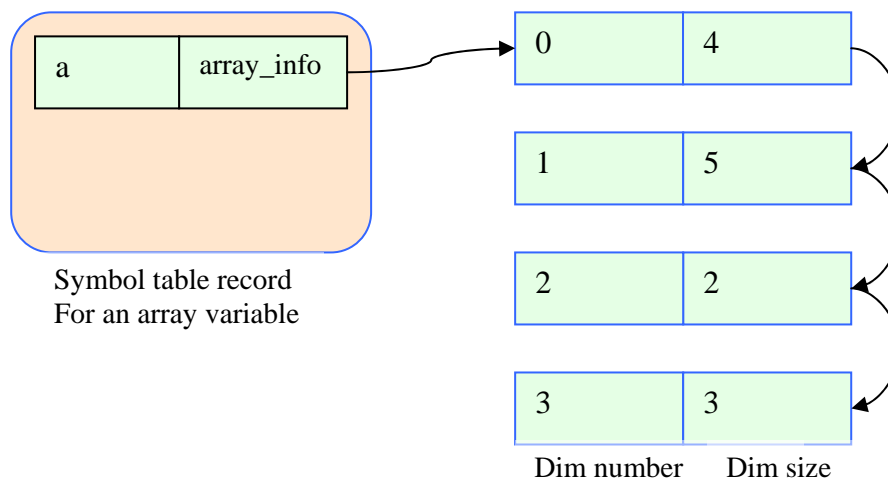
And thus this what this variable single elements looks like:

Array element	Signature list type	Signature list content
a[1][2][1][1]	int	101
a[1][1][1]	Int*	101 107
a[1][1]	Int**	101 107 107
a[1]	Int***	101 107 107 107
a	Int****	101 107 107 107 107

thus array or pointer to same type are the same.

As for data access it's not safe and done by shifting from current pointer.

And this is how this array variable looks like inside the symbol table record



We made this convention to make run time array variable allocation possible, if compile time array allocation was the only thing wanted it would been easier and just knowing : number of dimensions and each dimension size then an we can allocate one straight forward block of memory and any element would been just a shift from the start address.

How ever this will not allow the case where an array is defined like this:

```
int a[2][];
```

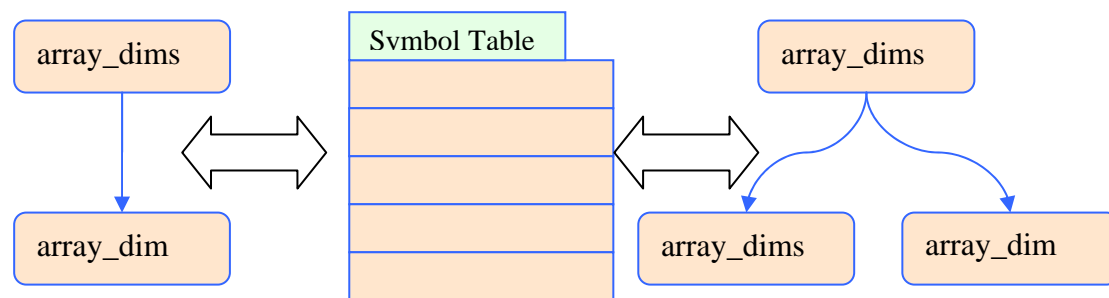
which we could later do this operation onto:

```
    a[0]=new int[10]
```

```
    a[1]= new int[5]
```

which could been done by the convention we presented above, besides type checking would be faster too.

(in compiler design in general, there is no restriction on how to implement arrays or anything, every one does it his way)



Above is the node that holds array dimensions inside the AST

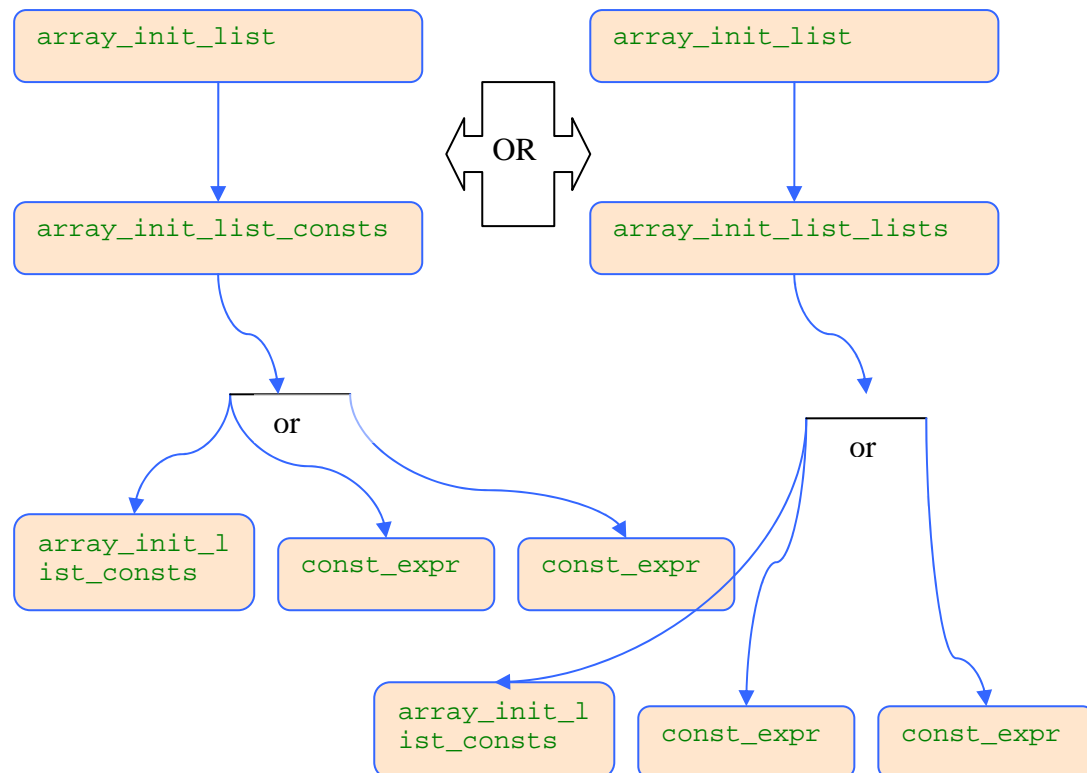
An array_dims node has two possible shapes, one with one an array_dim node and one with array_dims node and array_dim node, which refers to the rules: one or more dimensions.

The check done against an array variables, is that:

- all dimension info must be const integral data
- add the array variable to the symbol table if no other has reserved the name before
- for each dimension add a new DimInfo object to the array info list
- update the signature list of the record by appending more pointer levels.

As for **array initialization list handling:**

First we'll present the nodes to represent this list then we'll discuss the situation we have:



Child nodes are not deterministic and they resemble the grammar rule they represent, any way this way enables us to represent any complex initialization list of which ever depth and compatible with the C++ array initialization list rules as tested under VC++.

Laterally these rules can be expressed in:

```

int x[3][2]={ {0,1},{1,2},{0,0}};
int x[3][3]={ 1,2,3,4,5,6}; //fill 6 elems and the rest are 0s
int x[]={ 1,2,3}; //the dimension becomes 3
  
```

Array initialization rules:

```

int a[][][2]={ 1,1,1}; //error, one empty is allowed
int a[][2]={ 1,1,1}; //ok, set dim size to 3
int a[1][][][2]={ 1,1,1}; //error empty should be to the left
int a[][][]={ { 1,1,1},{1,1,1}}; //truly just one empty
int a[3][3]={ { 1,1,1},{1,1,1}}; //3 by 3 fill
int a[3][3]={ 1,1,1,1,1,1}; //all fill
int a[3][3]={ 1,1,1,1}; //left to right fill, rest are 0s
int a[2][3][2]={ {{1},{1},{1}},{{1},{1,2},{1,1}}}; //three list levels ok
  
```

- only one empty to the left.
- left to right fill.
- list contains count less or equal to dim_size.

list values must match the basic type(the last dim type) and no type conversion allowed basically this is how it works to match a list against dims:

recursively: **from current dimension** if current list is constants then it's count must be less or equal to the remaining elements in the array and that is:
count of such list is <= multiplication of cur and remaining dim_sizes

if list is list of lists then:

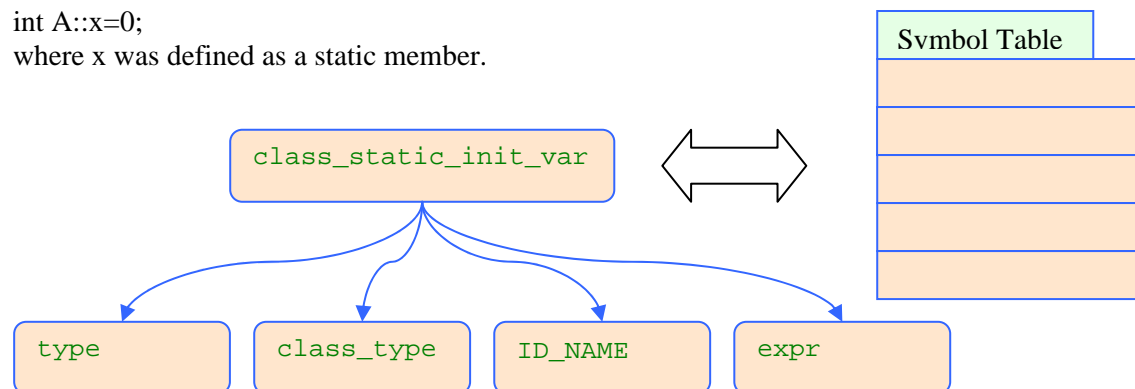
count of list must be less or equal to the current dimension size, then we check each partial list against the next dimension.

this was fully implemented in type checking, how ever in code generation we didn't go that far.

4-3-20 class static member initialization node:

`int A::x=0;`

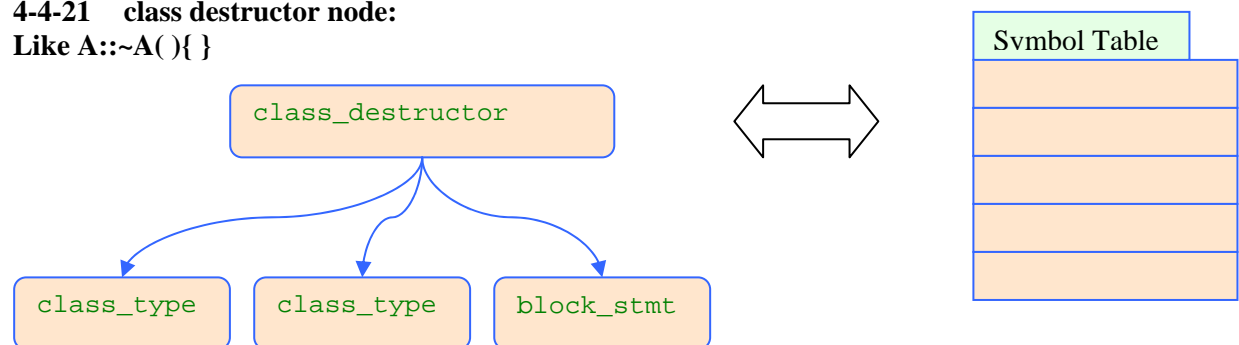
where x was defined as a static member.



- first of all we search the symbol table for the variable `ID_NAME` which has a parent of a class type.
- Check that type matches the signature of the record found
- We forward the check to the expression
- Type of expression must match that of the variable or could be implicitly converted to it.

4-4-21 class destructor node:

Like `A::~~A() { }`



For destructor implementation we do this:

- check that the second class type node matches a declared destructor inside the first node which is supposed to be a class type node.
- Of course if it matches then parameters don't exit anyway
- If declaration found then we update the symbol table reference node filed to reference this node as a definition node.
- We forward the check to the `block_stmt` node, but before that we create new scope rules by increasing depth and setting current parent to be this constructor

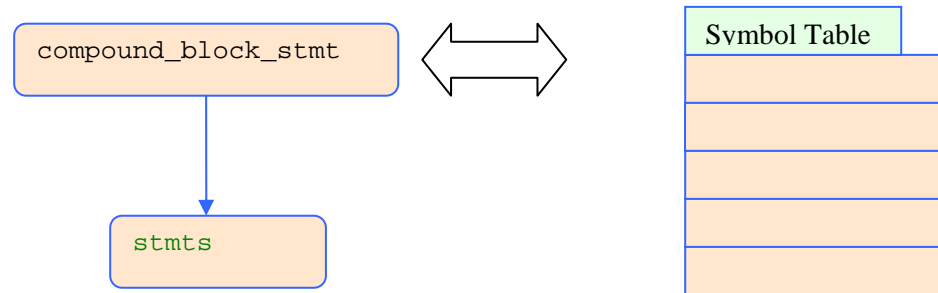
4-3-22 stand alone block statement:

i.e

```

{
    int x=0;
    {
        int x=0;
        { int x=0; }
    }
}

```

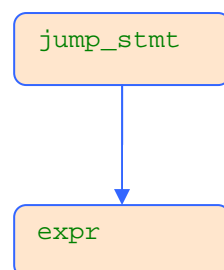


We allow the un named block statements inside { } :

- create a new scope rule by creating new block random named symbol table record
- increase depth and set parent to this created record
- forward check to the child of this record, which could be empty or stmts node
- after exiting we restore old scope info.

4-4-23 jump statements:

i.e break, continue; return 0;



Assuming the use **return** is valid within any block of info, the only check against the **return expr**; statement returned expression must match that of the parent block if available.

so we define a global variable to hold current possible return type and we match against it thus return variable is changed when a new function block is entered only.

so if current function return type was **int** and we faced a **return "ddd"**; statement then an error is issued

As for **break** and **continue**:

They are allowed to be used only inside loops, so what ever the deep this statement was nested within, what matters is that it has a parent node of type loop, to do this functionality we did this:

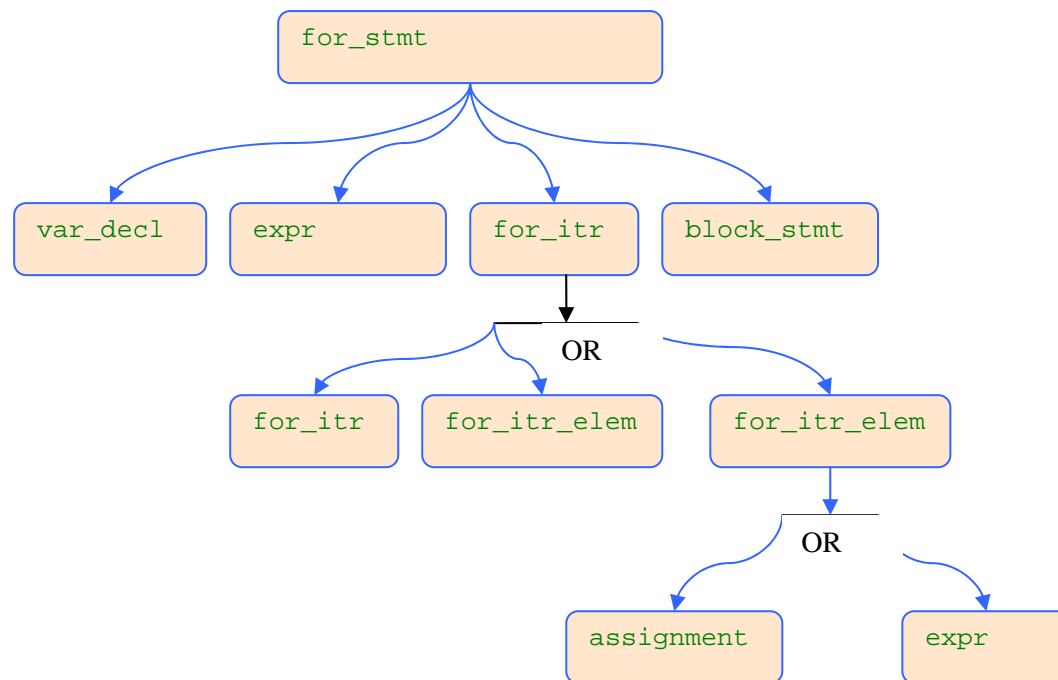
We defined a global variable named **loops** which is incremented each time we enter a loop body node and decremented when check of loop body is ended, so the value of this variable is always ≥ 0 .

When this value is 0 then no loops are around so use of BREAK CONTINUE is not allowed and error is issued.

4-3-24 for statement:

```
for(int i=0; i<10; i++){ ... }
```

```
for(int i=0, j=1; i<10; i++, j*=I, x++){ ... }
```



The two forms above are available.

- for statement check is by forwarding checking into it's child nodes only
- the variable declaration node is considered as the for initialization section just not to create another separate node for it.
- expr is any valid expression, we didn't put any conditions on it, since all expressions evaluates to a value even if they were pointers.
- for_itr node has two forms: either one for_itr_elem node or two nodes (one for_itr and the other for_itr_elem)
- so for iteration could have one or more elements as child nodes
- one for iteration element node could be an expression or an assignment, which either it was it will be checked for correct definition.

4-3-25 if statement:

i.e:

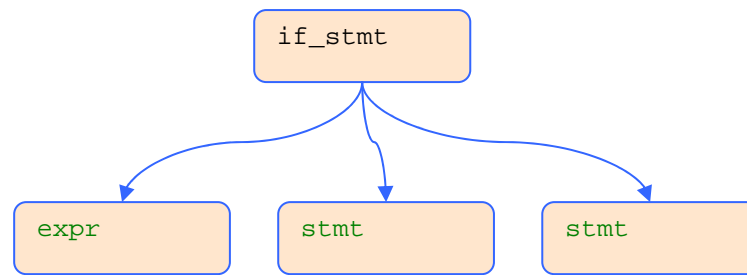
```

if ( x<5) { ... }
if( x==5){

}
else{

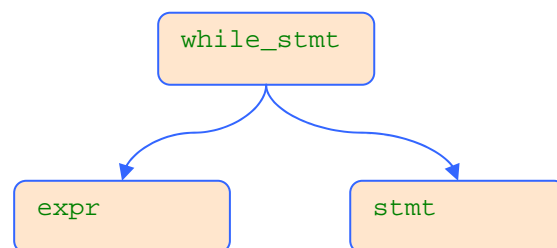
}

```



- check is forwarded to children

(26) while statement:

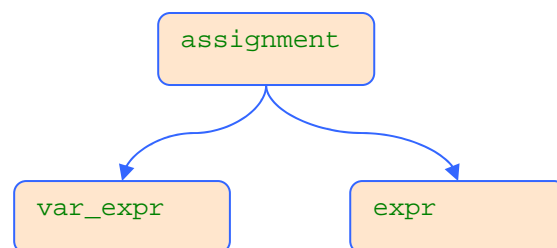


The same.

4-4-27 assignment node:

X=5; X+=5; x-=5; x*=5 x/=5;

All are considered assignment forms:



- the first node must be a valid variable expression form (variable, array element, member variable...)
- calculate the type of the second node
- match the left and right side types.
- Allow implicit cast of types.

Now we are gonna write down all valid expression forms and their type management:

We already defined an expression to be one of:

```

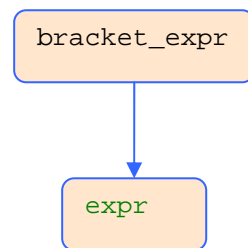
const_expr
| var_expr
| unary_expr
| binary_expr
| allocate_expr
| proc_call
| cast_expr
| '(' expr ')'
  
```

so, when expression is found in any node or any where, it's type will be calculated.

An expression is also a statement case, for example 5; is a dangling expression useless, yet still considered as a statement.

4-3-28 bracket expression:

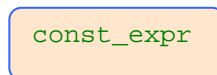
i.e: (2+3), (2*(3));



A useless node, but left to preserve program structure if we wanted to out put or print it.
Forward check into the lonely child.

4-3-29 constant expressions:

4, true, false, 'c', "dddd", 4.44



A leaf node with a value, so just push the signature.

The string constant is pushed as a pointer to a char type., further processing is done against it in code generation.

4-3-30 a variable expression nodes:

A variable expression is the one that can be assigned a value and stored in the symbol table. Variable expression have the most variant forms to accommodate as possible forms as possible.

These include:

Variable expression sample	Name
a[3][2][3]	Array expression
a[1][2].x	Array expression of objects that have member variables named x
a[2].x->y.z->u[0]	Too damn nested variable expression
a[2]->x	Array expression of object pointers
*x	Values of pointer x
* a[2]->x	The same as above
x	Normal variable
this	This pointer of current class
a.x->y.z	Object with nested members
A::x	Static variable
x->y->z->u	Object Pointers tell u
This->x.u	Members of this class pointer

All variable forms above are checked and considered ok by our project.
Next we'll mention some of the above cases as nodes in the AST

- when we got a variable called, then we search in the parent scope for it, if the parent scope was a class then we search in the base classes if it wasn't in his parent and so on till first scope, so we use the classes as upcoming parents.
- if the parent of such variable wasn't a class then we search recursively in the parents until the first scope too
- if parent was a class then we allow access to protected variables from parents and deny it if it was private.
- Thus we enable multi level inheritance

4-3-31 array expression node:

i.e:

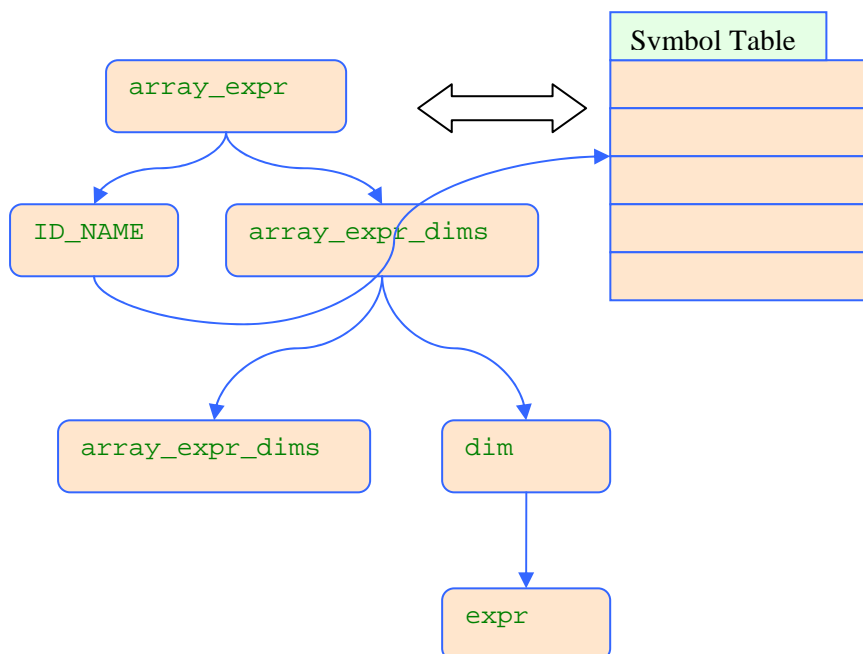
assuming `int a[2][3][4];`

then `a[2][3][4]` is a valid array expression node with type `int`

`a[2][3]` is a valid array expression of type `int*`

`a[2]` is a valid array expression of type `int**` (though this is never used)

`a` is not an array expression but it's ID and so it's place is not here.



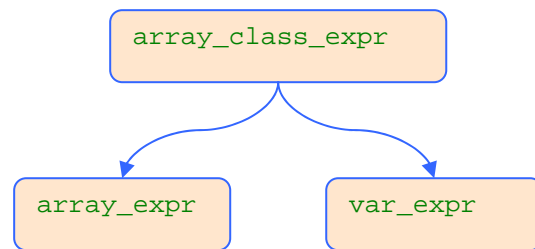
- get the symbol table record of this array ID, which must be declared before used and dimension is defined too.
- Check how many `[]` are there, total count must be less than the found array dimension count - 1
- Indexing must be integer
- No check on the index values, whether it exceeds the current dimension size or not.
- Type of this expression depends in the number of brackets `[]` mentioned and the type of the array.

4-3-32 array of objects expression:

Assuming: `A a[3];` where `A` is a class type, and initialization was done

If we got `A[1].x` then we need to check it out.

Here the node is like this:



- we forward the check to the first child, we talked about before, we check the second child
- the second child must be variable expression where it's first child is a member in the array expression to the left.
- The total type of this expression will be the type of the right side child.

4-3-33 array of object pointer:

Assuming: `A* a[3];` where A is a class type, and initialization was done

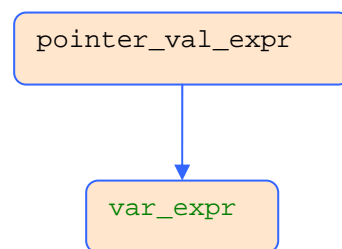
If we got `A[1]->x` then we need to check it out.

The same as above, and the return type is the same, the only difference is that left side must be of class pointer type.

(34) pointer-value expression:

Assuming `int* a=new int(1);`

Then `*a` is the value 1



We forward the check to only child.

The variable expression child must be a pointer type, that means: the last element in the signature list is 107 (pointer type)

The type of the total expression will be by removing the last element (the pointer flag)

4-3-34 variable expression node:

Int a;
Int* a;
A a;



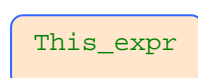
a is the identifier

ID_NAME

a leaf node with name of identifier.

- check that this identifier has already been declared.
- update node's signature list to match the symbol table record signature.

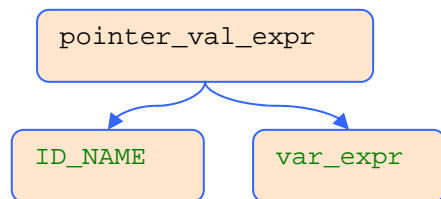
4-3-35 this expression node:



- this should only be used when current parent of the parent is a class type
- the type of expression returned is a pointer to the class type

4-3-36 class variable expression:

a.x; where a is a class object



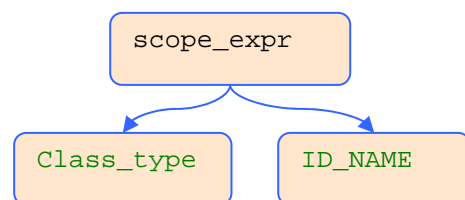
Left side must be a class object

Right child must be a variable expression with it's left child a member variable in the class.

The returned type is the type of the right child.

4-3-37 class static members:

A::x



Right child must be a member variable in the class that is the left child

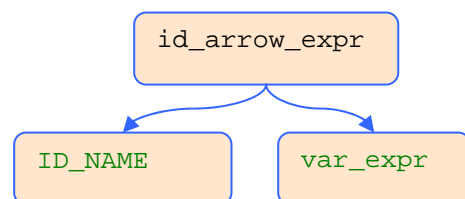
So we search for the class of name class_type inside the global scope

Then we search the ID_NAME inside the class found as a parent, this ID_NAME must be static member, and with an access privilege

The returned type is the type of the right child.

4-3-38 object pointer expression node:

a->x where a is an object pointer and x is member variable



Two child nodes

First search for the variable ID_NAME starting by current scope.

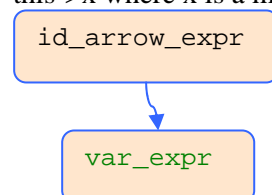
We check the var_expr node as a child of the object

Check if access is allowed.

The final type of the expression is the type of var_expr, the right child.

4-3-39 this pointer expression node:

this->x where x is a member variable

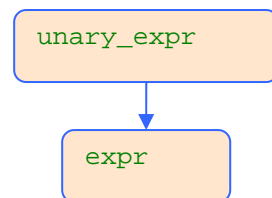


Here we got one `var_expr` child node, we check it's type, check if this is allowed to be used in current scope info, the returned type is the type of the right child.

4-3-40 unary expression nodes:

Like:

! x -x +x ++x --x
x++ x-- &x



- As seen above, the unary expression has one child node, as for the type of operation it is stored inside the node.
- The returned expression type is the same type as the child
- The only check made is forwarded to child to check it self and calculate it type
- The only exception to this rule is the referencing unary expression (& expr)

```

int x=0;
int* y=&x; //ok      &x is a pointer type expression
int& v=y; //error    v is int, y is pointer
int& v=x; //ok      v and x are int types
  
```

when we define a reference to a variable we define another name for the same variable nothingmore.

so we create a symbol table record and set a reference to the original variable that means (if data was separate from names we reference two names to the same data, if not, we define a field which indicates that this record is a reference and doesn't contain the real data.

When such reference is defined in the function arguments we create the symbol table as a reference too, how ever we set the reference record when the function is called if argument was a variable expression.

So to follow our way in type checking with least modification we do this:

When declaring a reference we create such a record, set a variable that indicates that it is a reference and set it's reference directly, as for it's type signature we set it as the type referenced, so when such a variable is used in type checking we check it as a mere type as int. How ever when evaluating the program we use the reference field and check this for every variable.

Computationally it takes the least time to execute among other methods which needs to check additional type called Reference type for example...

```
Int* x=new int(1.5); //ok
```

```
Int* x=&dval; //error
```

Only Pointers against the same types are compatible.

Const variables are considered constants which are replaced later by their direct value, and thus can't be referenced.

a reference must be done against volatile variables only not against constants.

Though we said we put it as a pointer type, when assignments are made and operands are checked we make sure that reference and pointer don't always match.

The symbol table record has a field to indicate that it is a reference variable record and thus, this record references the original record, so any change to the original record the reference can see it as long it references it.

4-3-41 binary expression nodes:

There is no direct node for this rule, though it exists in the grammar, in the AST the direct replacement for this is two nodes: one for math and the other for logical operations.

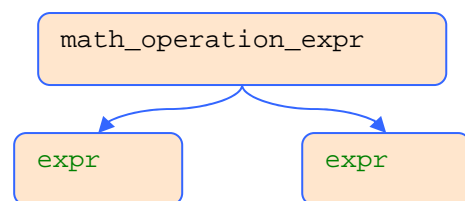
4-3-42 mathematical operations expression nodes:

`expr + expr`

`expr - expr`

`expr * expr`

`expr / expr`



- two expr child nodes
- we check both sides and calculate their type
- then we balance the two operands by checking implicit type conversion between the two sides.
- The returned type of the whole expression is the most common type if balancing was done.
- Balancing does this:
 - o If the two expressions have the same signature list then they are ok and type matches directly
 - o If not we check against simple types and see if conversion is allowed, like: if one is double and the other is one of (int, char, bool) then it's ok, if one is int then the other must be char or bool to allow the balance else an incompatible operand types is issued.

4-3-43 logical operations expression nodes:

`expr < expr`

`expr > expr`

`expr == expr`

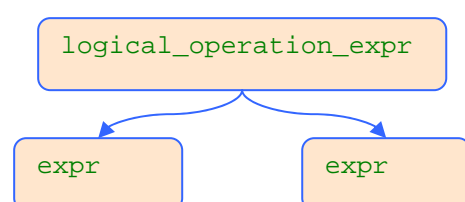
`expr != expr`

`expr <= expr`

`expr >= expr`

`expr && expr`

`expr || expr`

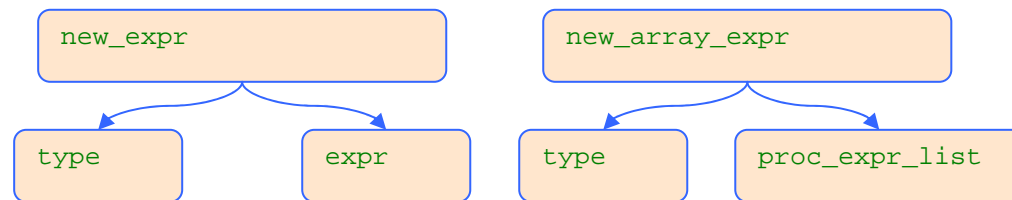


The same as math operations, and type is also a field inside this node that indicates which operation this node is.

Balancing of operands, the return type of this expression is a Boolean type.

4-3-44 allocate-new expression

```
new int[5];
new A(1,2,3);
```



As seen, new expression has two forms, and was separated into two nodes, both have two children.

The first form allows the creation of single dimension array for example:

Assuming : `int a[2][1];`

At some point we could do this: `a[1]=new int[5];`

Or `int* x=new int[10]` since one dimension array is compatible with a pointer to the same type of that of the array.

In this case the check made:

- calculate the type of `expr`
- type of `expr` must be a constant integer
- the returned type of the expression is a pointer to the type

the second new case is to allocate a class object like:

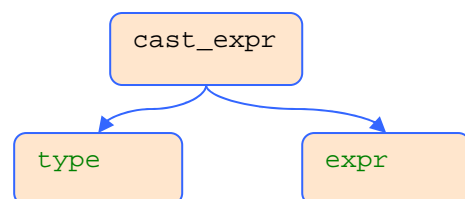
`new A(1,2,3)` which could be inside a context like `A* a=new A(1,2,3)`

the check made here will be like this:

- search the symbol table for a constructor with the same name as the type and the same signature generated the second child which is a parameters list
- returned type is pointer to class type

4-3-45 cast expression node:

```
(int) (x+y)
```



Casting an expression does the same work as balancing two operands of a binary operation, since we only provide simple type conversion only.

So we calculate the right child type and check if conversion is possible with the left side.

And finally:

4-3-46 procedure call nodes:

The only forms to call a procedure in our grammar is like this:

```
f (1,2);
a.f(1,2);
```

```
this->f(1,2);
```

```
a->f(1,2);
```

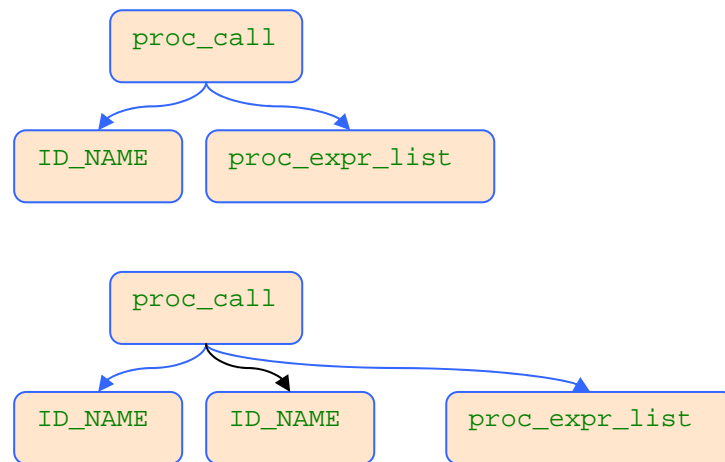
```
A::f(1,2);
```

in static functions

no nested variables with depth of more than two are allowed to call a procedure.

the right thing is to define the proc call as `var_expr '(' proc_expr_list ')'`

but to ease our job we did it as mentioned before.



The node has the previous two forms, more info about a node is found in the main node fields

- a call must be done against an existing declaration
- the argument list signature must match that of declaration
- access to the function must be allowed(public private, protected) these access info is found in the symbol table record.
- If scope dots were used then the function must be static

so for all four cases, the function signature is created out of the procedure call parameters and with finding the proper parent of the function we can find the declaration record if it's there in the symbol table.

The first form of procedure call, the parent is know, since we define functions in global scope, how ever in the second form, we need to find the class the has the name of the first child and lives in the global scope too.

When.

Most functionality that repeat between node checking are gathered in separate function to ease the job, we'll try to mention these function if we got time to write the class diagram of this project.

Basically this is as far as we could remember doing in type checking the program, next we'll talk about Code optimization.

5- Code Optimization:

Code optimization has more than one phase: one before code generation by fine tuning the AST and another is done during code generation, also after code generation by focusing on this operation it self and no other.

Optimization transforms the AST, in the sense that some nodes could become exposable, how ever this operation should never alter the program semantics.

At this stage we'll talk about code optimization done in the phase before code generation is done.

When we end code generation we'll mention the optimized methods use in the code generation phase.

(naming in next cases is our own invention)

5-1 Eliminating dangling expressions:

We can say that an expression is a dangling one when it has no effect in the program whether it's there or not, in our grammar this would be a statement like this `expr; :`

`2+3;`

`x*y;`

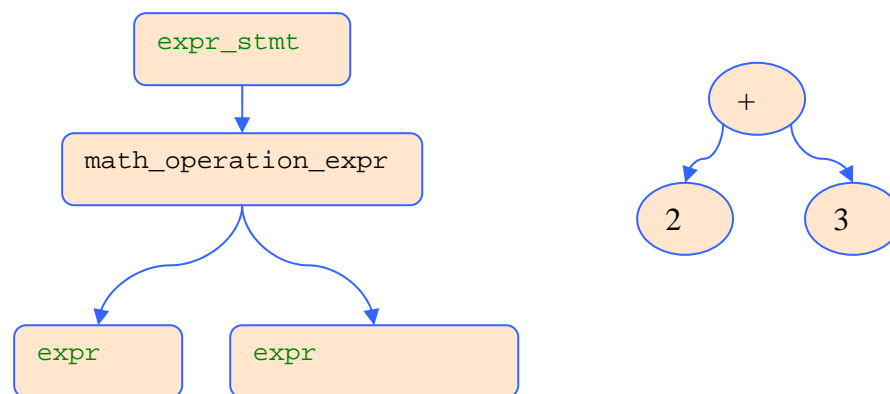
`!x;`

`X;`

`This;`

`This->x + 4; etc.`

which is allowed in our grammar, for example:



the above node is totally exposable, no matter how complex it is, and really complex it could be it doesn't affect the program what so ever.

How ever not all expressions that forms a statement are exposable, since some forms like: Procedure call are necessary to get called and they always exist in the form `expr ;` where `expr` is `proc_call`

Also the unary operations: left and right increment and decrement is excluded from the rule above

As for assignments like:

`X+2=0;`

They are not allowed and you get a type check error, since we only allow the left side of assignment operation to be a variable expression not any expression.

In a node (TreeRecord class object) we define a variable which is defaulted to true, which indicates that the current node will have code generated for it.

So we need not to alter the AST structure for a dangling node, we just set this variable to false and it will be skipped by code generation.

5-2 deleting unused variables:

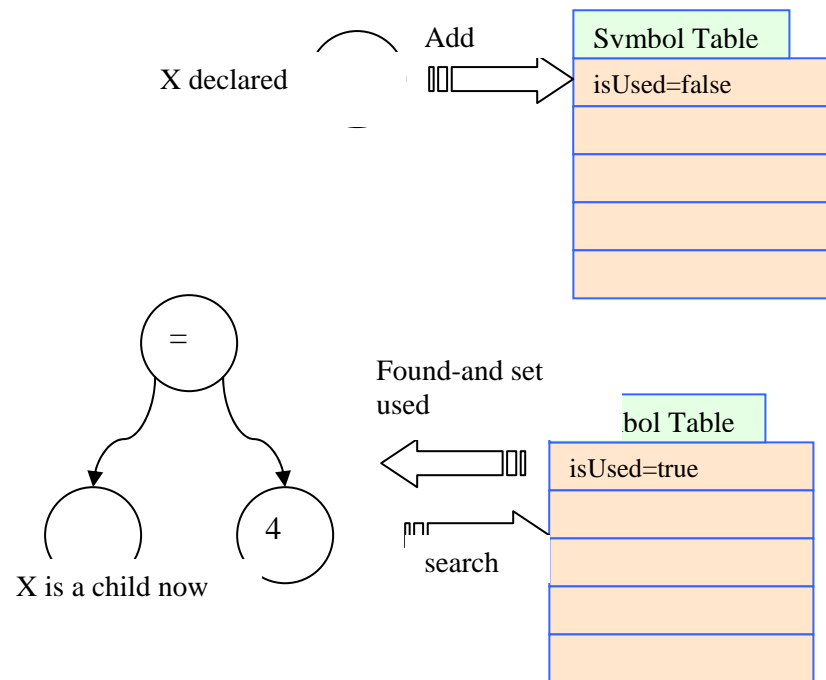
As we said before, we need not to alter the tree structure, just mark a node to prevent code generation for it.

Be default, when a variable is declared, it's marked UNUSED.

So Symbol table record have a Boolean variable isUsed, which is defaulted to false, to tell that this variable was used or not.

The way we do this is easy:

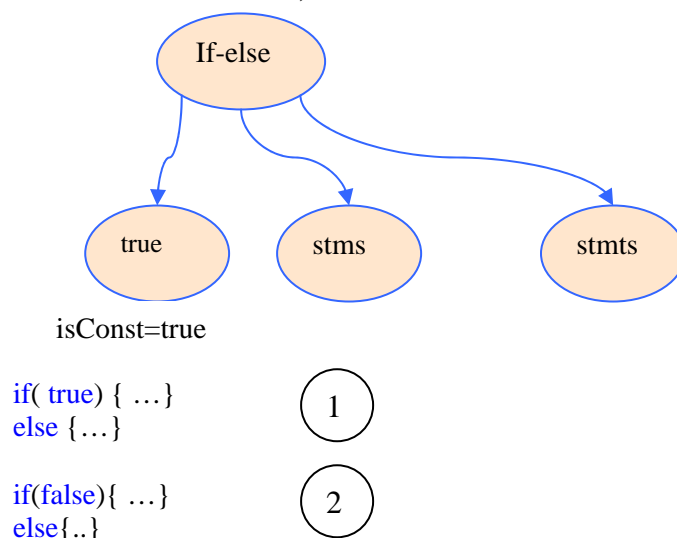
Any variables that is considered an effective variable, is one which gets searched for, so where ever this variables was used it will be called during type checking, this call for marks the record to be used (searched for and thus used)



This mechanism needs no special structure to achieve the functionality above.

How ever, the flaw in this approach: when this variable was called from within a dangling expression, which will set this variable to true, and the dangling expression is marked not to generate code for, but our variable has already been marked used, and code will be generated for it, this case is unlikely to happen but it might, any way, code optimization will never be able to make the program optimum, it just makes it better.

5-3 deterministic if, if else statements:



if(true)
if(false)

3

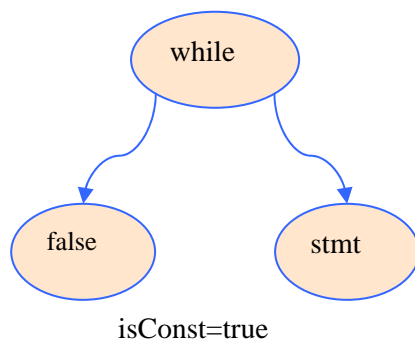
statements like if, if-else needs one expression to evaluate and enter the it's branch, so incases where this expression was constant, and could be evaluated at compile time during AST traversal, then we can tell which branch this if statement will enter, and thus eliminate a great part of code.

Above the first case, we'll always enter the first branch, so need for the else branch at all, and we mark it as dull, so when code is generated, we generate no code for the if nor the else and directly generate the first statements of if.

The second case: the same as the first, but here we generate code only for the else

The third case: the whole statement is ignored and marked dull when it's false or we just generate code for the child statement and not for the if when it's true.

5-4 deterministic loop statements:



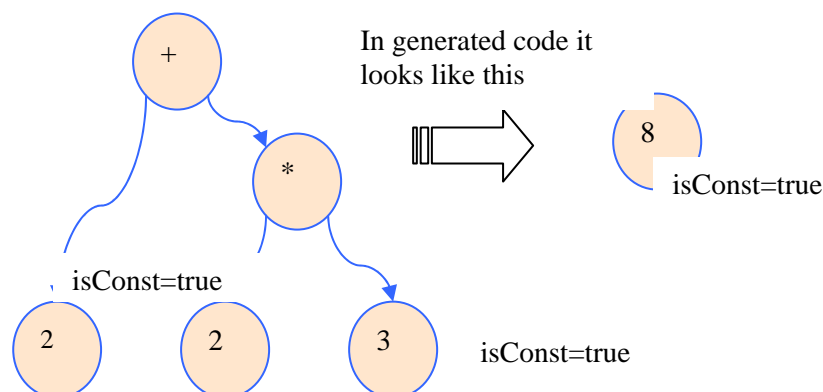
Same as if-else how ever this state is entered only when the condition is constant and false so we mark the whole node as dull and generate no code for.

5-5 constant expressions calculations:

one of the most common obvious code optimization techniques, which tries to compute all possible constant operations at compiler time.

Example:

$2+2*3$ could be calculated at compile time as replaced directly by 8



As we said before, inside a node we define a Boolean variable name isConst to indicate the current node is a constant expression.

At type checking phase, constants are leaf nodes, and at the end, we calculate and check the children nodes before current node is marked as done.

So for expressions, when the children are marked as constants, we do the operation on them, set in the current node, and mark current node as constant.

At code generation phase, when we face a node that is marked as constant (isConst=true) we generate code of constant type and stop with no further investigation on children nodes.

Another optimization on constants could be by considering the monadic of operation or special numbers in general, for example:

$1 * \text{expr} = \text{expr}$

$0 * \text{expr} = 0$

$0 + \text{expr} = \text{expr}$

So in this case, if we got a binary operation, we check the two operands if one is constant and has a value of 0 or 1 and depending on the current operation we mark the expression as a constant (only when multiplying by 0)

If it was multiplied by 1 then the code generated must be the same as the other operand, and this case is handled in code generation.

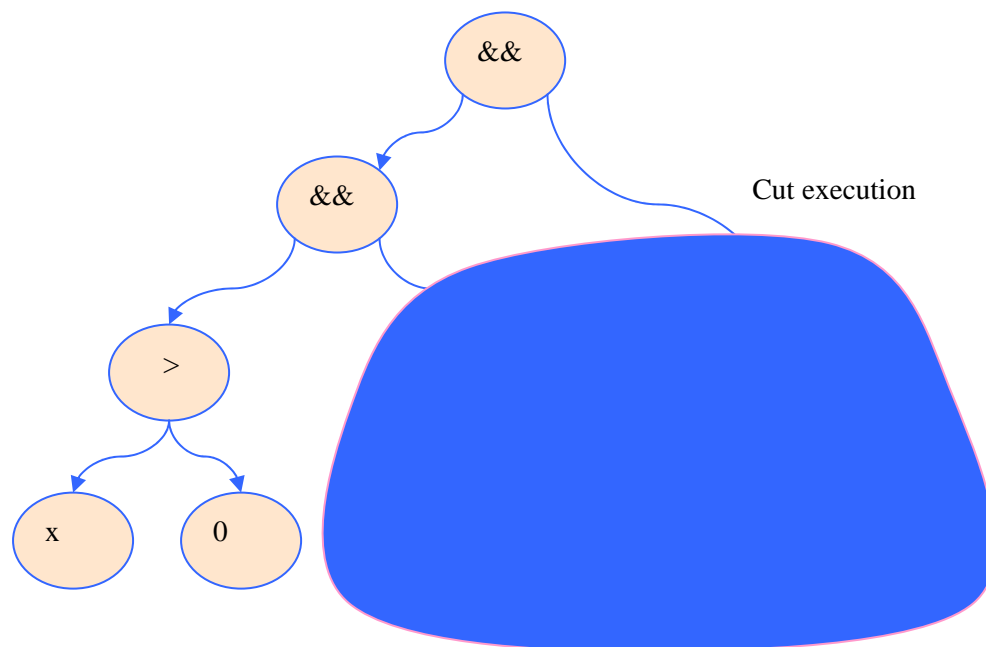
All above optimizations tries mainly to eliminate part of the generated code to speed things up, how ever other types of optimization are available that speeds up execution and increases performance at runtime, we'll mention one which we already implemented and generated code for, and another one which we are likely to generate code for before the dead line.

5-6 sequence of ANDs-ORs cut:

when handling logical expressions like

$(x > 0) \&\& (x < 0) \&\& (y == 3)$

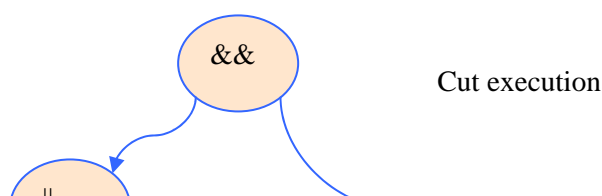
And Or tree



You can call it AND-OR tree cut, but at run time

So if the first logical node evaluated false, then there is no need to evaluate the other nodes, it's waste of time.

$((x > 0) \parallel (y < 0)) \&\& (z == 3)$



If $(x > 0)$ evaluated to true then there will be no need to evaluate consecutive Ors and we do an OR cut.

AND rule: At the failure of any condition the remaining conditions are not evaluated.

OR rule: the success of any condition stops the evaluation of others.

The above idea optimization was implemented and tested in code generation (we could see the results through the graphical VM)

We'll come to mention it again in code generation section, and show the code for it.

Finally, we'll mention an optimization technique which is **we didn't implemented yet**:

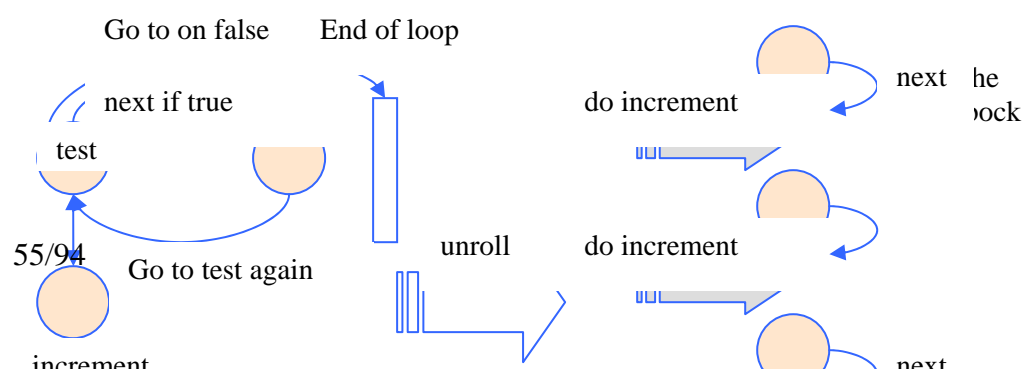
5-7 loop expansion (or loop unrolling):

`for(int i=0; i<10; i++){ ... }` the same apply for while

if we knew that this loop would only be entered for few reasonable time (say like 20 times) then we unroll the one statement (which is mapped into one block of code and few jumps and tests) into a consecutive statements.

This can be done both in the AST without code generation knowledge, by adding the proper nodes, or in code generation by unrolling it there.

Consecutive statement execution is much faster than that of test and go execution.

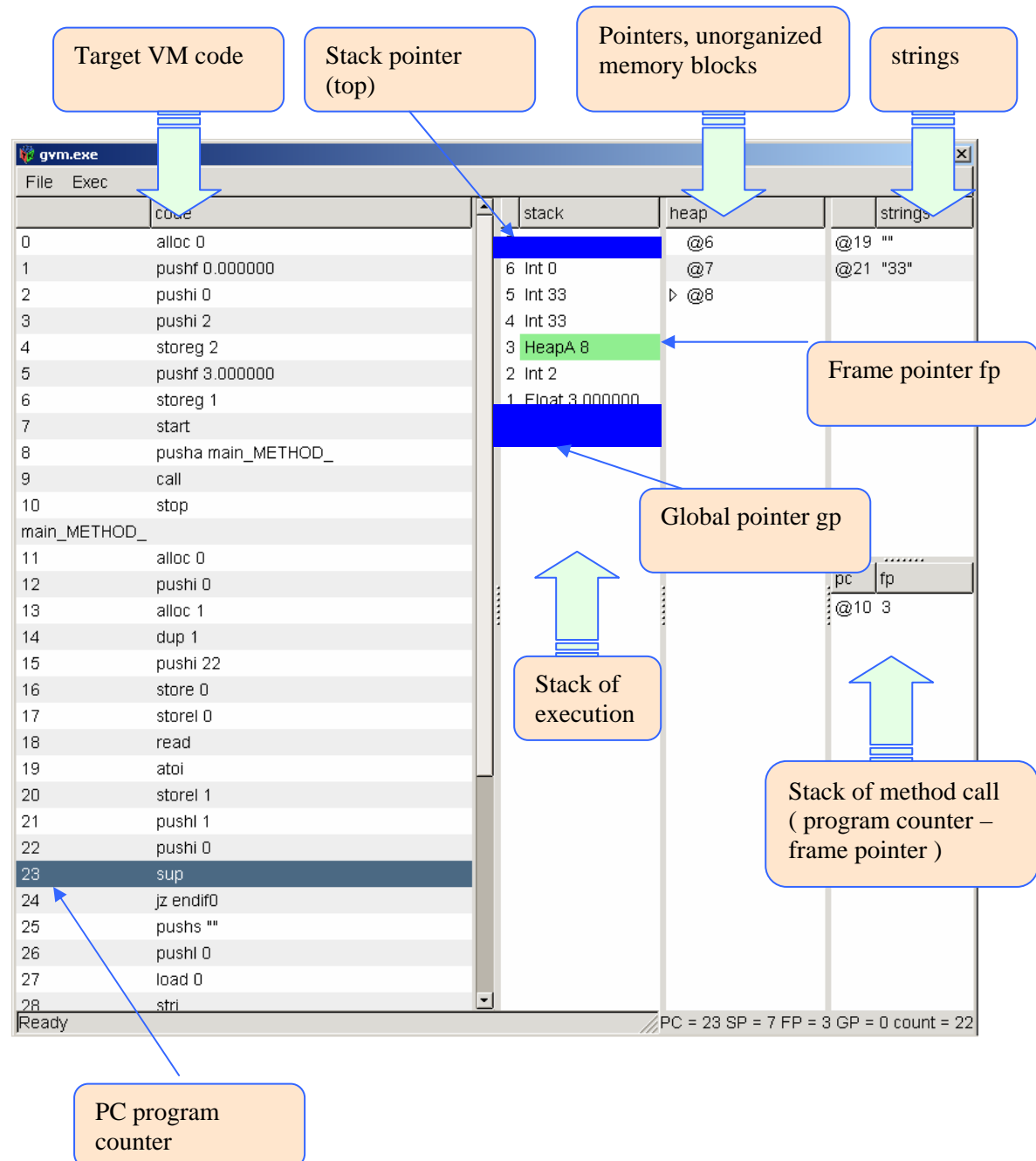


That's all for Code Optimization.
Next Code Generation.

6- Code Generation:

So we got the AST, done some optimization, our structure is ready and has enough information to start code generation phase, which in our case will produce a VM code to be implemented.

We want to say that code generation was the easiest part, but it wasn't, it took much time than we thought, but it was far more easier and less time consuming than type checking. Most of this phase was about well understanding the way the target VM runs, how to transform our AST into a sequence of statements in a sequential file and to make run in appropriate way.



The above figure is for the code bellow [just as a sample of what's going on]

```
int globalX=2;
double globalY=3.0;

int main()
{
    int localX;
    int* intPtr=new int(22);
    cin>>localX;
57/94 if(localX>0){
        cout<<localX<<*intPtr;
    }
    return 0;
}
```

Our VM has the above structure, and we need to transform our AST into that sequence to the left, to be able to execute properly on the right.

We'll come to mention, almost every node in the AST, but first we start by the initialization steps:

6-1 Code initialization:

First of all, we only allow code generation after the program was checked and contains no errors what so ever.

- allocate first stack position for NULL values
- initialize global variables and update them with their assigned values
- generate the start command
- allocate space for main return value
- call the main method call
- generate the stop command
- generate code for the whole tree now.

6-2 Variable Storage: and retrieval introduction:

You can say, we consider three types of variables:

- global variables
- local function variables
- and class member variables

we differentiate between the three above in both storage and retrieval

so to tell a variable position or address in the current execution stack, we store it's shift from a well known pointer which we can call and retrieve any time, and the only pointers known are

- frame pointer : fp
- stack pointer :sp
- global pointer: gp

so global variables are stored relative to the GP

local function variables are stored relative to FP

and runtime allocated variables (like newly allocated pointers or classes) are initially allocated relative to stack pointer and later saved in a non volatile pointer or address.

So the Symbol table record now has two additional fields:

- shift
- shift_from

the shift field is integer value indicating the position of this variable relative to the relating execution pointer

shift from takes one of three values:

- 0 relative to the global scope
- 1 relative to a function scope
- 2 relative to a class scope

Variable allocation depends both on the AST and Symbol table info

So for global variables, we do this:

- global variables are stored in a prog_decl nodes and var_decl nodes

- when a global variable declaration node is found, a global shift value is incremented and the right variable storage command is issued (like Pushi for integers, PUSHF when double, ALLOC Size when classes, arrays , pointers ...) all these are discussed in more depth later
- the initial storage command will be to reserve a place in the stack, nothing more, so it contains the default value of the variable (like 0s for consts, the gp[0] for NULL pointers, and the size of class data for objects ...)
- after initialization is done, we update these values with their assigned expressions if they have any

the above process is done first before any thing in the program

as for function variables or class member variables, they are allocated, initialized, updated when the relevant node is found.

6-3 complex variable initialization:

class objects:

tell the size necessary to store this class with all it's data and it's parent data and issue an allocate command.

Object Classes are stored just like pointer classes, and needs a delete command to free it's memory

pointers:

initialized to NULL by pushing the dedicated address for NULLs (the first address in gp that is gp[0]) which initially has the size of 0, so trying to access information from a NULL pointer will cause unknown error results, that is a vm implementation specification thing.

arrays:

if array size is not known at compile time, then it will be like pointer initialization, if it's dimensions are known then we reserve a block of size equal to the multiplication of it's dimensions.

This doesn't match our type checking idea, but if we got time, we'll do it.

6-4 variable retrieval:

simple types variables:

simple typed variables are retrieved by simple pushg, pushp command

complex typed variables:

class variables:

also like simple types using pushg, pushp which will push the address of that class

class member variables:

push the class address and load the value at the proper shift

arrays

also like simple types using pushg, pushp which will push the address of that array

array elements

push the array address and load the value at the proper shift

pointers

also like simple variables using pushg, pushp which will push the address of that pointer

all above operations are detailed further each when proper node is discussed.

Next we traverse the tree and generate the code for each node that is marked with generateCode=true.

6-5 AST nodes Code generation details:

After initialization was done for the global variables, we generate code for the remaining nodes.

6-5-1 program declaration nodes:

Out of these nodes we only implement the definition nodes: that mean any declaration will not have a matching code.

These definitions are:

- class definition
- function definition
- class member function definition
- class constructor
- class destructor

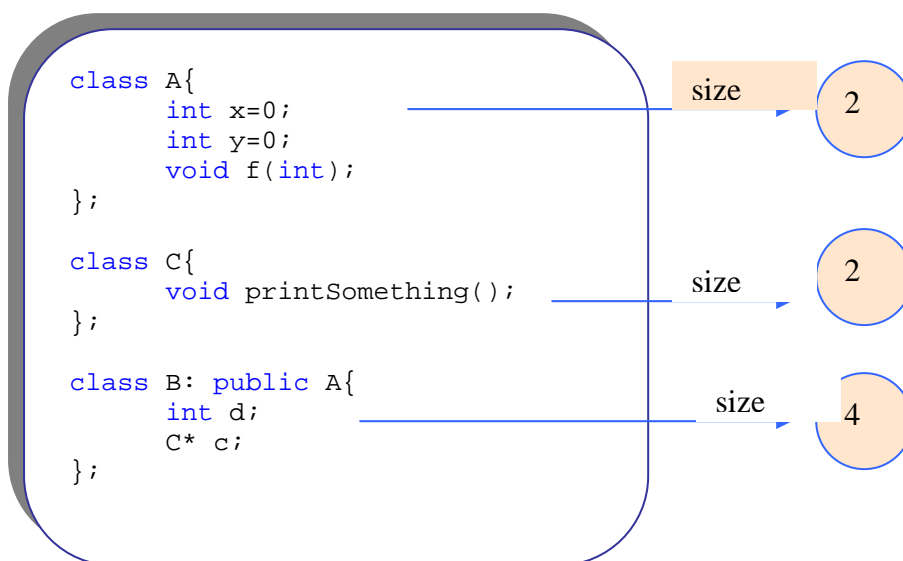
6-5-2 class definition node:

when we see a class definition we just calculate it's size required for it's allocation command: the size is equal to member variable sum of sizes plus parent sizes

so: since we store objects as an organized block of memory just like pointers, then no internal variable declaration is to have generated code for, even internal function declaration, since we already said that only definitions will have code generated for the.

Class size calculation is a dedicated function which updates the class definition node which we added for it a new field called size:

For example:



When seeing the class A definition node, we update it's size with 2: two member variables it has, so it has size of two

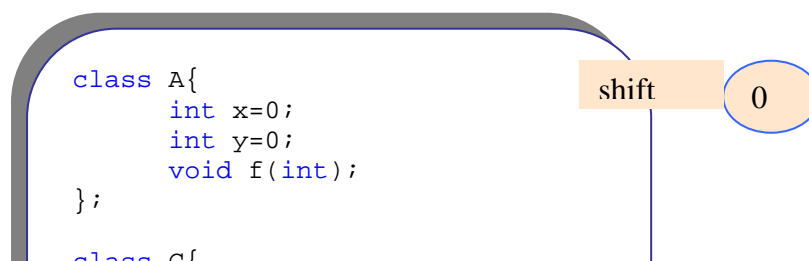
Class C has no size at all

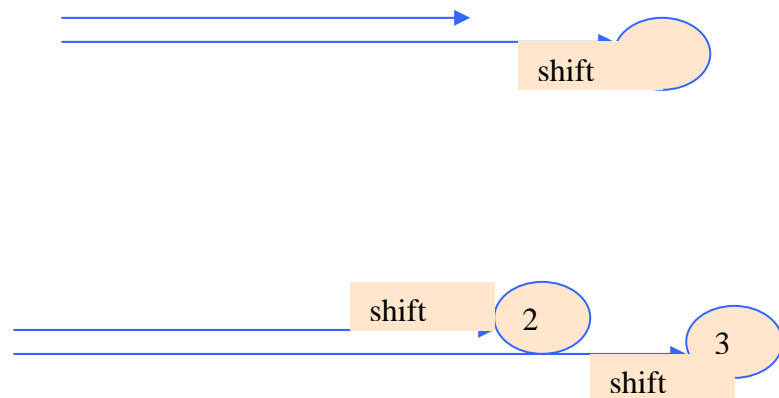
Class B: it's member variables are two and thus it's local size is 2, it has one parent A with size 2 also, so total size will be for class B.

This size is enough information to allocate space for a class

In addition to calculating the size of this node, we also calculate the member variable shift from the 0 start location of the class, this information is vital to load and store values in member variables

For example, the above classes





Shift from=2

Shift of derived class members variables begins after the sum of sizes of base classes (whether it was one or many, it doesn't matter, the rules still applies)

This shift and shift_from is updated into these variables symbol tables records.

This process allows us to do this:

If we got a variable declaration like this:

A a;

Then we issue a command like:

Alloc 4;

And when we got something like this:

a.d then we load the 3d location in a address [2d index].

So when we see class definition node, we generate no code for it, but we do some calculations to make objects and members of that objects storable and retrievable, these info are used in other nodes.

Construction and destruction is discussed later

6-5-3 function definition code:

6-5-3-1 Unique labeling (naming) generation:

Of course, a program might have many functions of the same name but different parameters (function overloading) so each one must have a unique name to be called later without conflicting with others.

Simply we define a char* field in both the node and symbol table record to hold a name of the function, which is the string name of it appended by a static number incremented each time used.

So when we find a function call node, we reference it to the function definition node which has that name, and function call code generation is issued properly.

The labeling is not exclusive to function calls or definitions, even for, while, if statements... uses labels which has got to be unique, how ever, since these kind of statement are used once in the program, so we can either depend on the unique name generation and storing them in the nodes or just generate them at the time needed and free them when we are done with them (almost at the same moment), so they are volatile names

```
int function(int x, int y){
    cout<<x<<y;
    return 0;
}
```

_function_function_0:

```
pushi 0
pushi 0
pushl -1
storel 0
    } Init params
    } Get passed values
```

Above is a simple function example:

Steps to generate a function definition code:

- generate the function name code
- generate function parameters code
 - o initialize shift value (a new function)
 - o generate initialization code for every parameter
 - o update parameters count
 - o assign passed arguments to the initialized parameters
- generate the block_stmt node code
- generate a return code

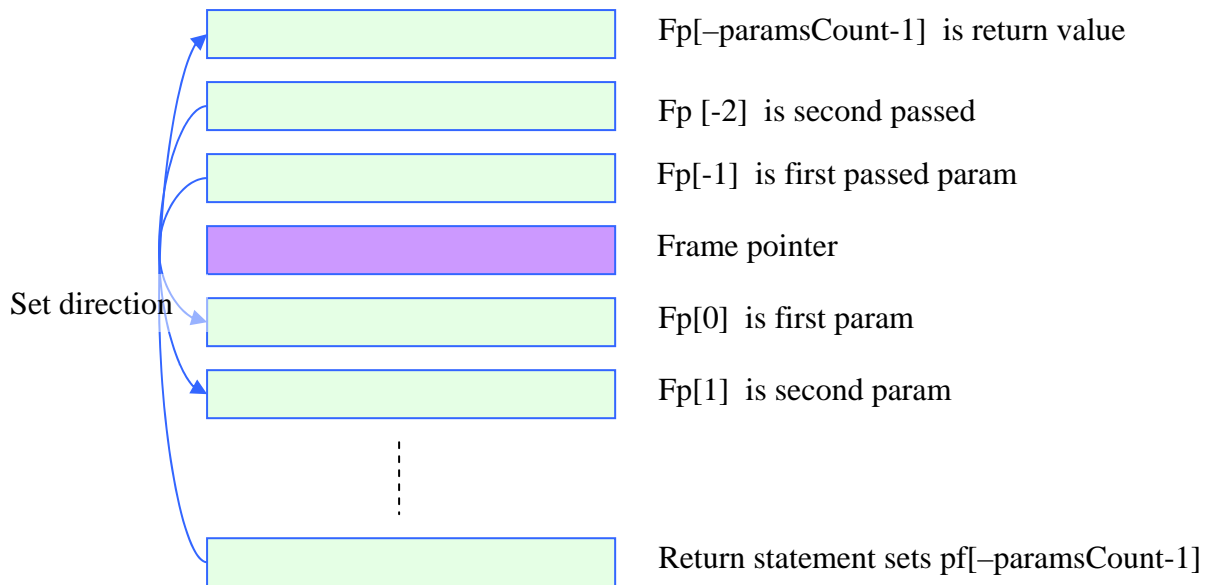
assuming passed parameter values are passed before the frame pointer fp we assign such stacked parameters to the variables of the local initialized params
we assume returned value is one step before the parameters.
All these assignments are done relative to the current fp:

This code formation is done to match the procedure call code which we choose to have this convention:

- push an initialized return value [reserve a place for return value]
- Push passed parameters onto the stack
- Call the procedure

So if a pointer was passed as a parameters what happens:

- first, it's address is loaded and put on the stack as parameters
- when calling the procedure this param is stored inside the proper local param
- so any change to the local param which is it self an address affects the real data



This is how the execution stack will look like for a function call.

6-5-4 class function definition code:

For a class function definition, we depend on the same mechanism as ordinary functions, but with a sole difference that we pass an address to the class before the **function call**, so all passed parameters becomes shifted 1, and that needs to be considered in the **definition**. So by taking this change into consideration the shift rule becomes:

Parameter position shift inside the function = shift

Passed parameter values shift before the pf = -shift -2

Where for normal function it was: -shift -1

The name of the class function will append the class name for clarity of reading nothing more:

For example `A::f (...)` will have a name like `CLASS_A_f_23`

6-5-5 class constructor definition code:

It's like a normal class definition but with few additions to allow inheritance action:

The form we allowed for inheritance is one parent class, so no multiple inheritances allowed however depth of inheritance is not limited.

6-5-6 class destructor definition code:

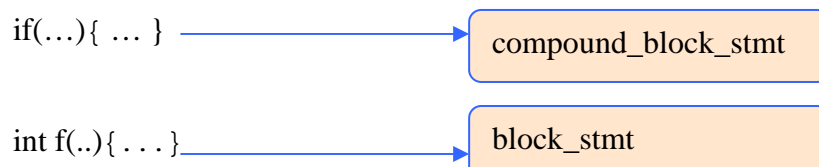
Same as a class function definition code generation, normally a destructor should be called when scope is left or when delete operation is issued.

Like constructor, successive destructors are called when inheritance is specified in the class definition.

Next we show code generation steps for the statements children:

6-5-7 block statements:

We differentiated between normal blocks and function blocks: for example

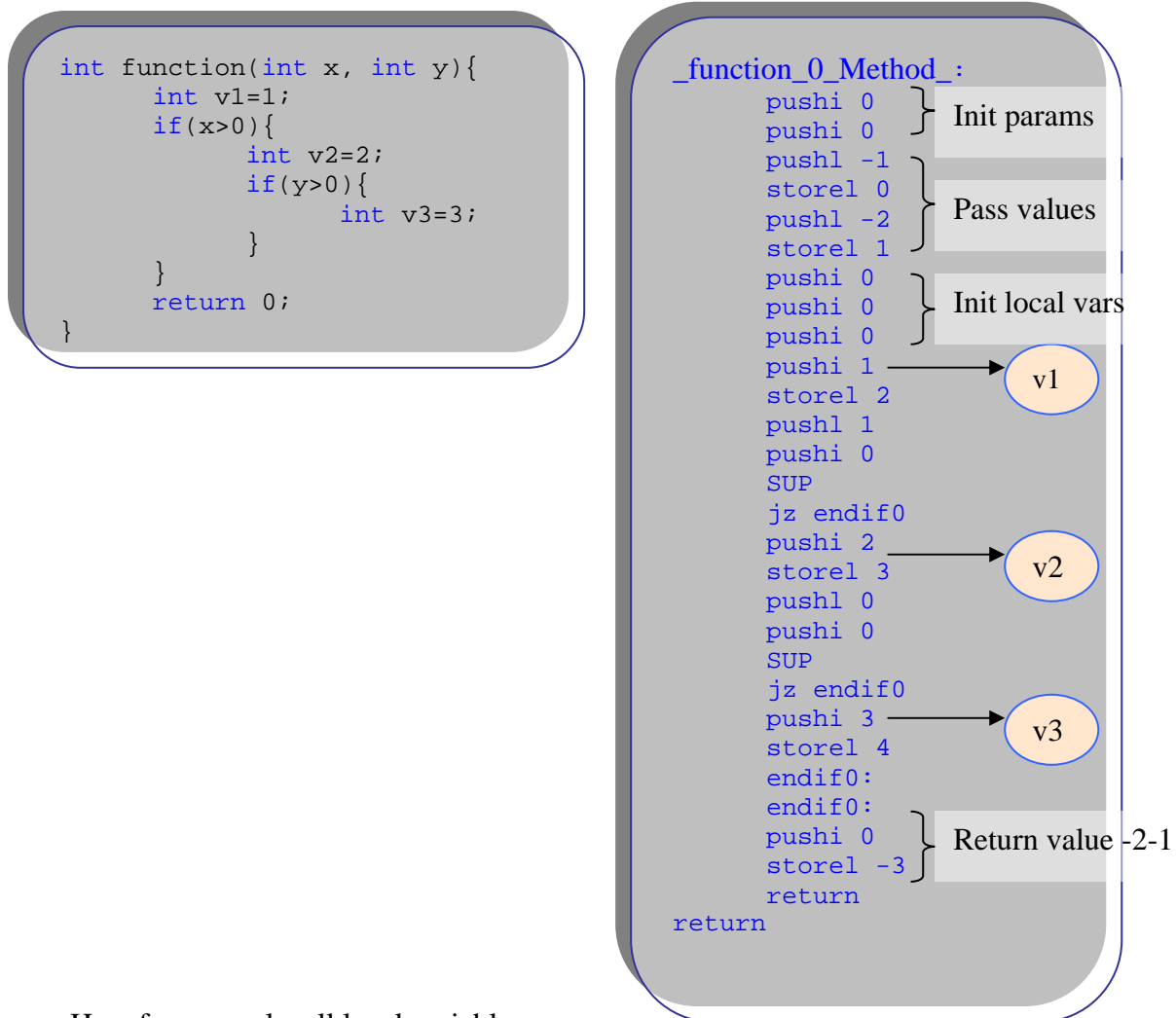


the compound block statement is just one node above block statement which is a dummy node to tell that the block is not the function body nothing more.

we made this convention because of the following fact:

when we see a function definition we initialize all the variables inside the function body, by making a place for them starting from the fp.

Where ever the variable declaration was we need to know it's storage location relative to a known pointer to be able to get it, so if we got this situation:



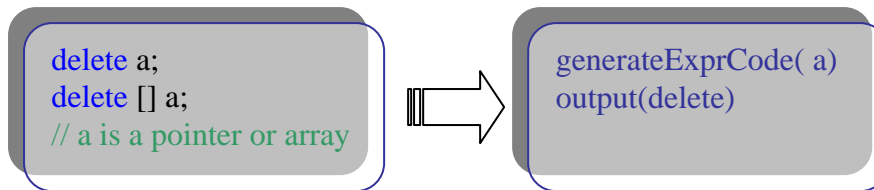
Here for example, all local variables even

In nested blocks are stored in the beginning after the parameters (when setting the passed values they will disappear from the stack and local variables will be directly next to parameter locations as soon as it's done.

So after local variable initialization of the block we generate the remaining statements, by passing the command to child nodes.

6-5-8 delete statement:

Delete is necessary for allocated data, this includes: deleting pointers and arrays.



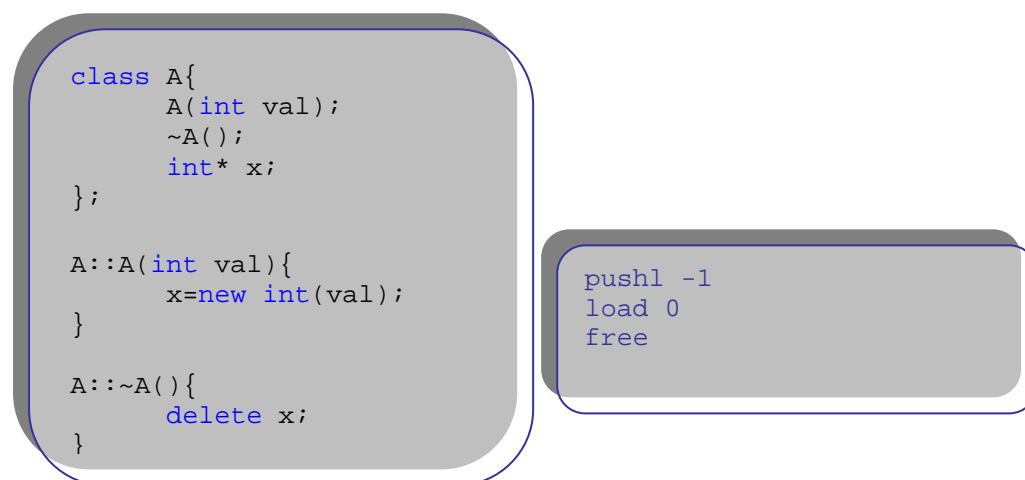
First we generate the a code, which in this case is an address, push it on the stack then issue a delete command.

In cases where this is a complex variable, the rule still applies for example:

Delete a.x; where x is a pointer inside the object a.

here a address is pushed first, then x is loaded by loading the shift, then delete is issued.

GenerateExprCode method, is one which is capable of generation of nested variables expression like the one above.



for example the above piece of script, deletes a local member pointer, it shows how to load the address of itself from parent object and free the variable, here object is pushed before the constructor function call.

6-5-9 expression statements code generation:

As we said before in optimization, we don't generate code for expression statements except the procedure call and part of the unary operations.

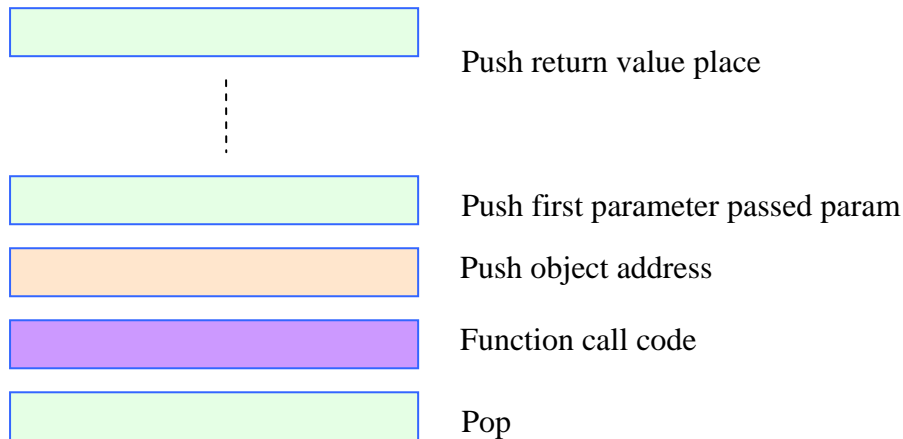
6-6-10 procedure call statement code:

We talked before about this topic when we came to mention function definition code generation.

So we'll just describe the process:

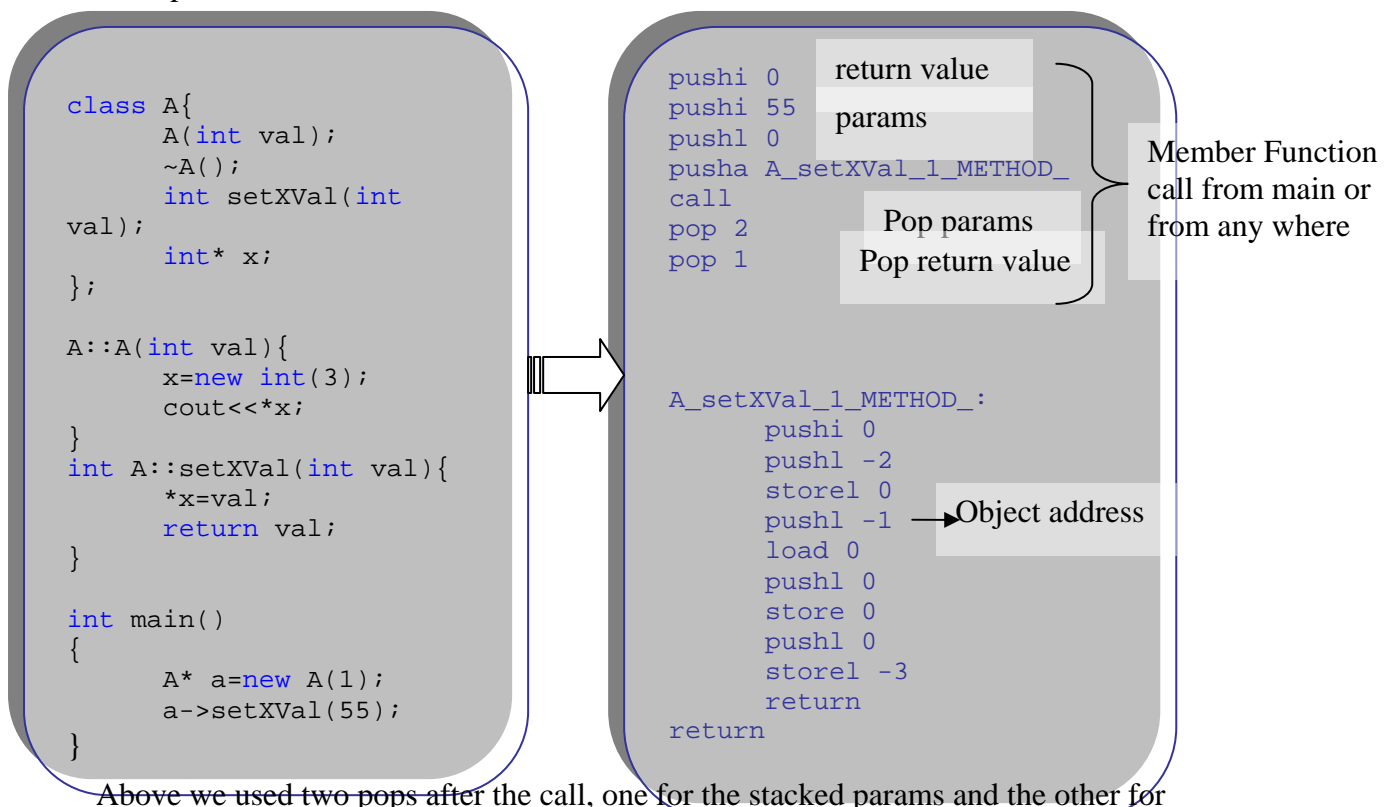
- for a normal function call:
 - o generate code for return value, that pushes the value onto the stack
 - o generate code for the parameters, which are pushed onto the stack so a function definition finds them there when he looks for them

- make a function call command with the same name of the definition
- pop the stacked parameters
- pop return value, since an expression statement then the return have no use for us.
- for a class member function call:
 - the same operations as above with a sole difference:
 - after pushing the passed parameters, we push the address of the object just before the function call



This is how the statement stack will look like (not execution stack)

Example of member function call:



Above we used two pops after the call, one for the stacked params and the other for the return values, this is because, when using the procedure call as a part of an

expression (not a statement) this return value is required to remain on the stack so in that case only the params are popped.

6-5-11 unary operation statement generation:

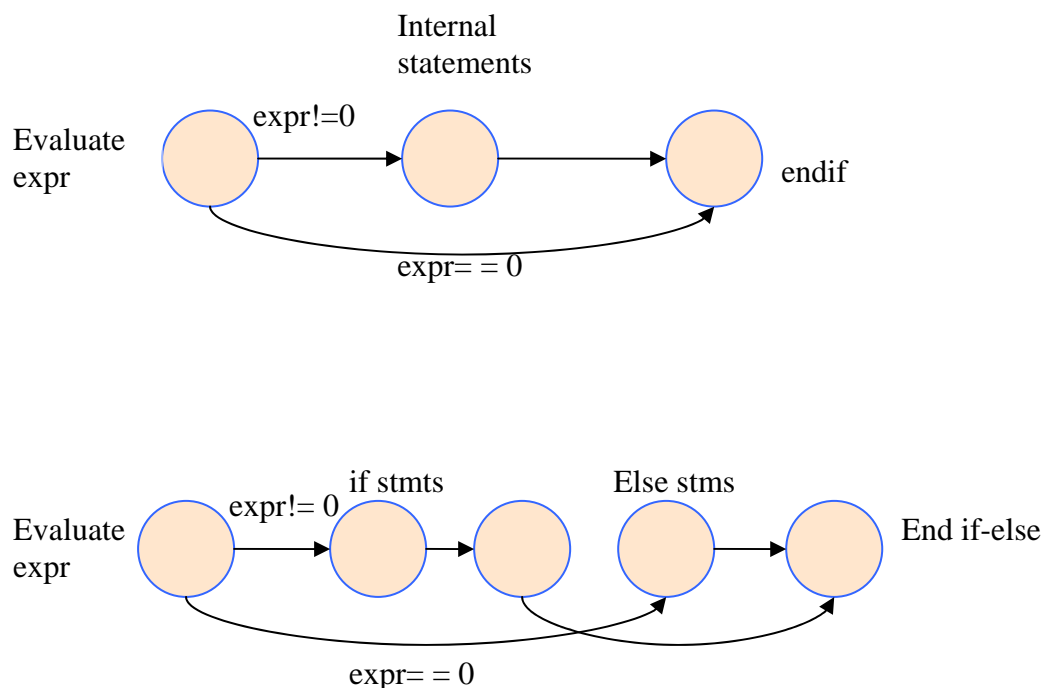
```
++X;
--X;
X++;
X--;
```

these 4 are [the only] considered statements
we'll discuss them when generating all unary expression codes.

6-5-12 if statement code:

First if with no else: if(expr) stmt

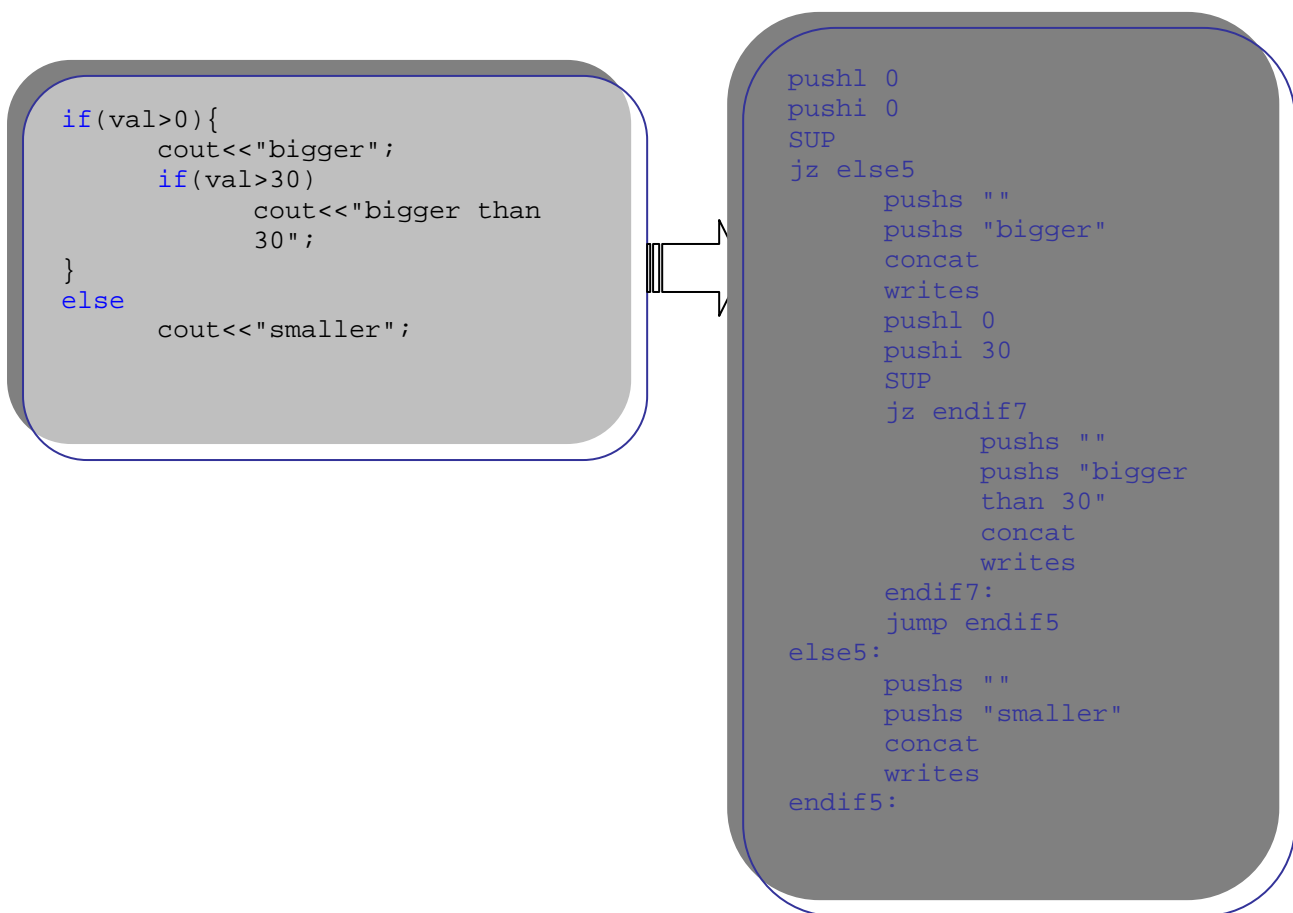
- generate the conditional expression to push it on the stack
- generate end of this if statement unique label
- evaluate, if 0 then jump to that endif label
- generate the statement code



When else is there:

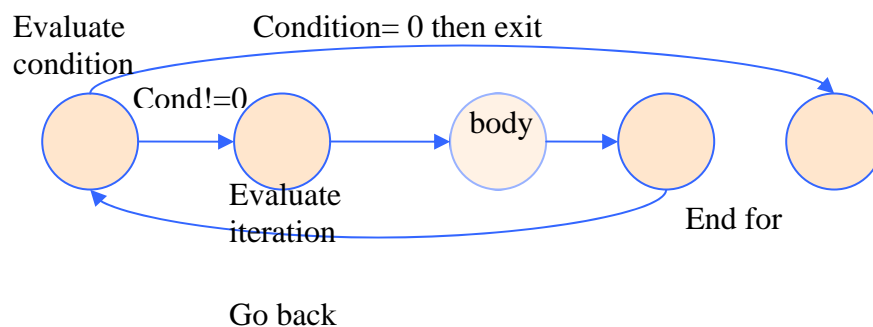
- generate the conditional expression to push it on the stack
- generate end of this if statement unique label
- generate start of else label
- generate condition code, evaluate, if 0 then jump to that else label
- if not 0 then continue
- generate the if body statement code
- output the jump to end of if-else statement

- output the start of else statement
- generate the else body
- output end of if-else



No matter how nested the structs were, labeling should never miss

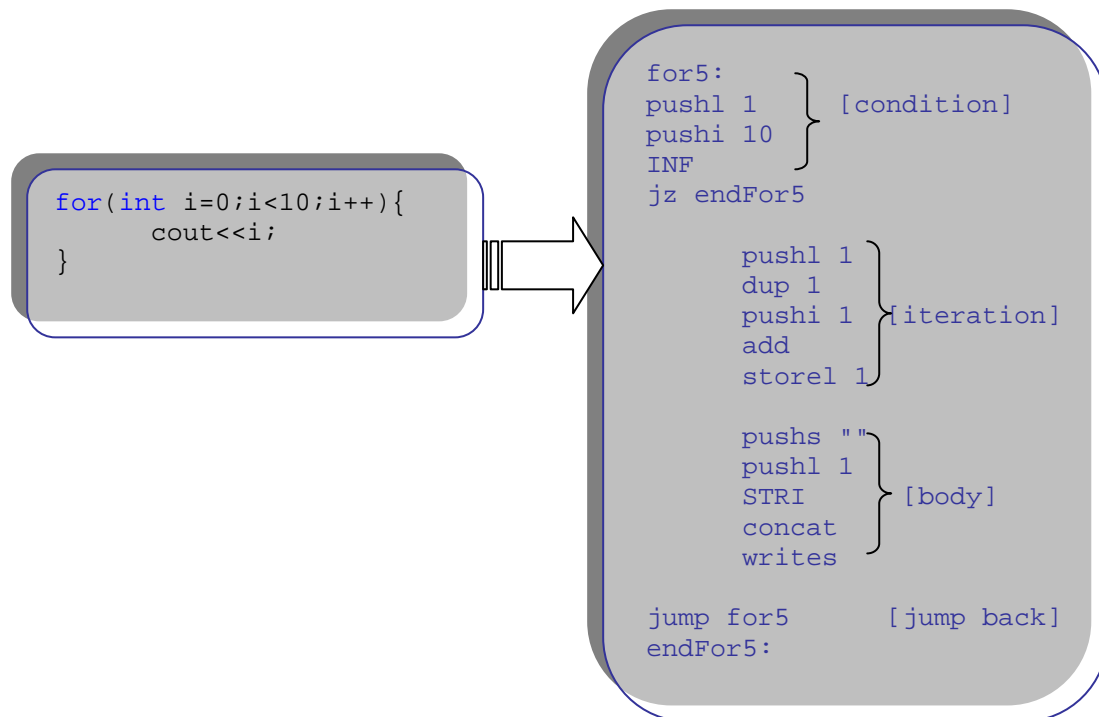
6-5-13 for statement code:



Steps for code generation:

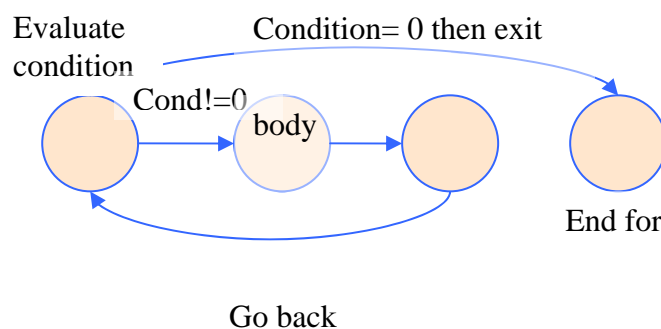
- generate for and end of for labels
- output for label

- generate condition expression code, evaluate condition, output jump on 0 command to end for label
- generate iteration expression code
- set current start-end loop labels for Continue-Break statements
- generate body code
- set current start-end loop labels again for Continue-Break statements incase they were changed
- generate jump statement to the for label
- output the end for label

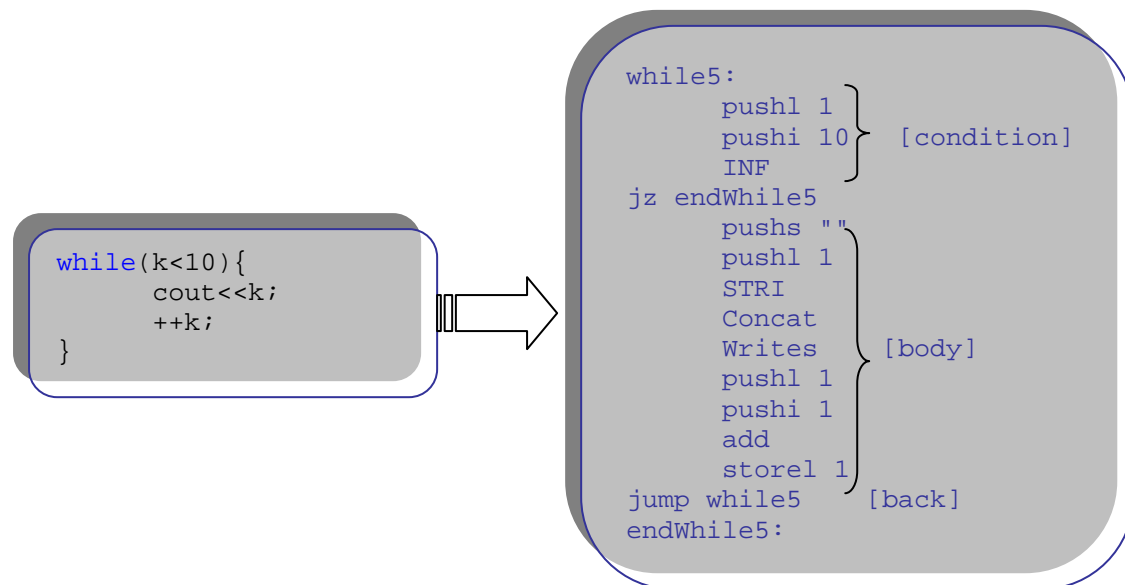


The variable declared inside the for statement is already set relative to the frame pointer like any other variable declaration.

6-5-14 while statement code:



Just like for but without an iteration.



6-5-14 jump statement code:

Break, continue, return

return expr;

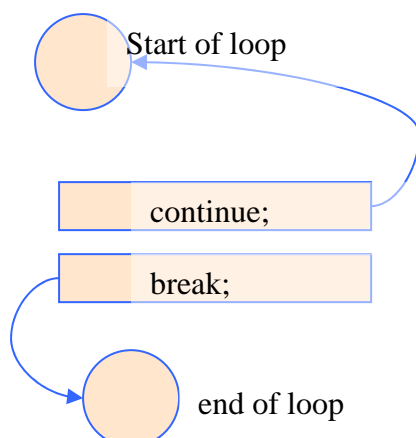
as mentioned before, return values of a function is stored directly behind the parameters of function and which in turn are behind the frame pointer and thus, the previous command does this:

- generate the expression code
- store it back in the position `fp[-params_count-1]` if this function is a normal function and `fp[-params_count-2]` if it was a class method

normal return statement without expression, just issues a return code.

Continue-Break: we define a global variables to hold the current start of loop [whether it was for or while] also the end of current loop.

If this loop has internal loops too, these labels are restored when they are done to the previous loop labels (recursive methods)



```

while(k<10){
    ++k;
    if(k==5)
        continue;

    for(int i=0;i<10;i++){
        cout<<"internal loop="<<i;
        if(i==3)
            break;
    }
    cout<<"    external loop="<<k;
}

```

```

while5:
    pushl 1
    pushi 10
    INF
jz endwhile5
    pushl 1
    pushi 1
    add
    storel 1
    pushl 1
    pushi 5
    equal
    jz endif7
        jump while5
endif7:
    pushi 0
    storel 2
for9:
    pushl 2
    pushi 10
    INF
jz endfor9
    pushl 2
    dup 1
    pushi 1
    add
    storel 2
    pushs ""
    pushl 2
    STRI
    concat
    pushs "internal
loop="
    concat
    writes

    pushl 2
    pushi 3
    equal
    jz endif11
        jump endfor9
endif11:
    jump for9
endfor9:
    pushs ""
    pushl 1
    STRI
    concat
    pushs "    external loop="
    concat
    writes
    jump while5
endwhile5:

```

6-5-16 assignment and variable storage retrieval code:

5 types for assignment:

Var_expr = expr; [normal assignment]

Var_expr +=expr;

Var_expr -=expr;

Var_expr *=expr;

Var_expr /=expr;

In all cases we need to put the address of the var_expr on the stack before operating (if it's a complex type like array element)

- load the variable expression (left child) address
- generate right side expression code (push onto stack)
- generate casting code for the right side expression if it needs any
- for mathematical assignments only:
 - o generate the left side expression code and push it onto stack
 - o generate cast code for the left side expression
 - o generate math operation code between left and right side
- store what's on top of the stack back into the variable expression.

so we come to mention the loading of variable expression address:

they are too many cases which we'll mention few:

for example:

direct variables like x:

- if x is a simple type then no address is loaded for it
- if x is a complex type it's ok, we load it's address which initially has these three cases:
 - o if the symbol table record shift_from == 0 then we issue a PUSHG for the shift of that record, this is a global variable
 - o if the symbol table record shift_from == 1 then we issue a PUSHHL for the shift of that record, this is a local variable
 - o if the symbol table record shift_from == 2 then this is an object member variable called from within class function so we issue a PUSHHL -1 which pushes the object address then we recursively make load command for this variable.

It's a little bit complicated and consist of few cooperative function to load all variable properly for example:

If we got this form:

[x->y.z->n] and we want to load it's address so we can store value in it we do this:

We separate this load process into two different forms:

- the first loads x address which uses the precious three case scenario
- then we recursively issue load commands on the right sides of this expression, which are all marked with shift_from=2 and a shift value telling their location in current memory block, so such a variable turns to become something like this:

```
pushl 2
load 0
load 0
```


load 1
thus we get the address.

For arrays: we don't load the single element address, since only the array it self has an address, as for elements they have values.

How ever when we want to store in the array element we do this:

- generate the array variable address in the same previous way, even if it's nested
- issue the proper store command no matter how nested it was

for example:

a[3][2]=3; where a is define as int a[4][4]

this turns to become

```
pushf 0    [ a address ]
generateExpr ( 3*4 + 2)    [ the element position ]
push 3     [ the value]
store
```

so we can say there are two kinds of loading we operate on:

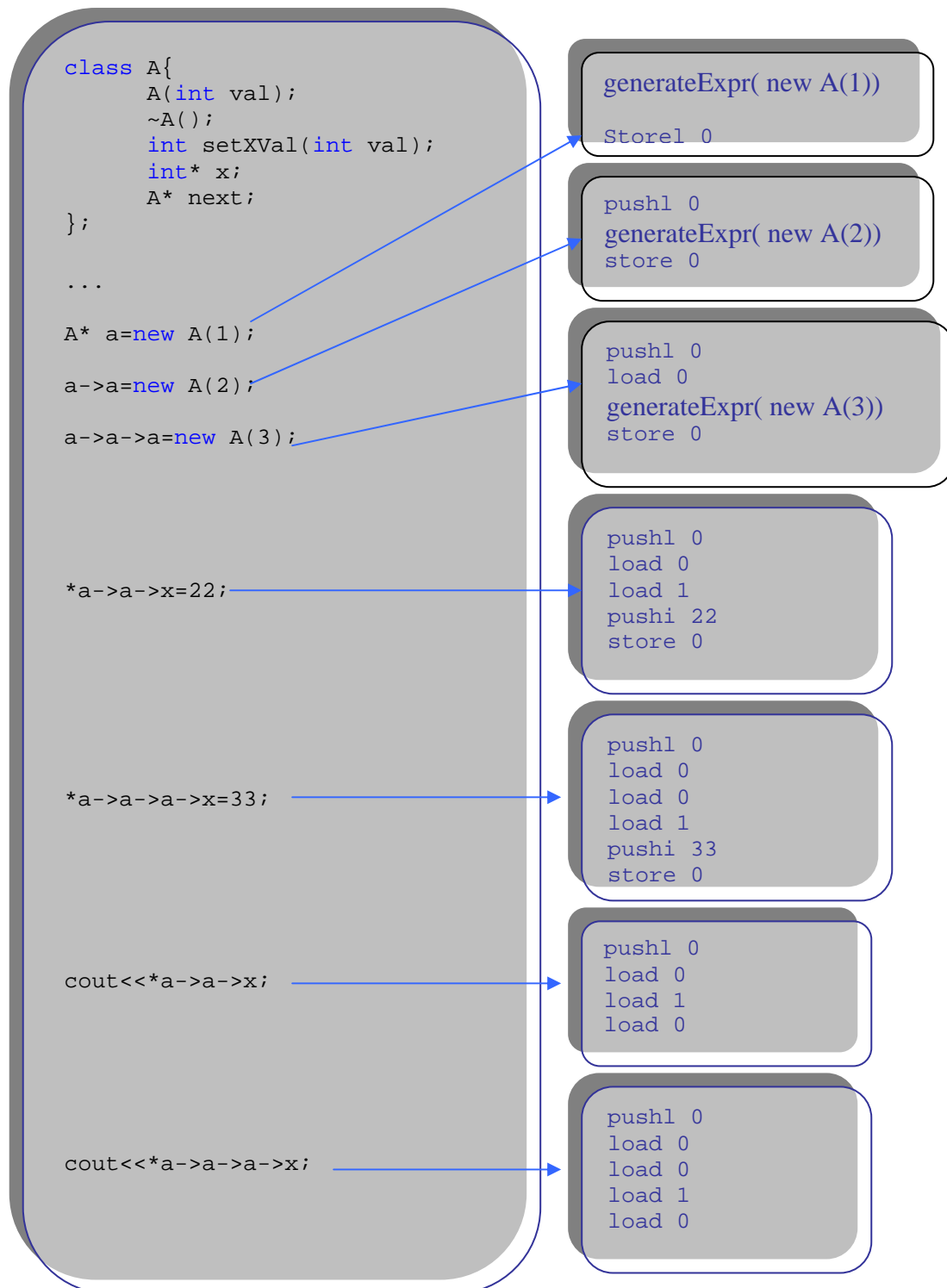
one to prepare the loaded for storage

and the other to prepare it for direct usage

the first case is one level deeper in storage point of view.

Another example:

Assuming:



This load-store recursive base method works on all variable forms, or example the above turns to become a linked list.

Of course if the loaded address doesn't exist or NULL the vm will issue a run time error and terminate.

Since we are talking about variable storage and retrieval, we'll mention how objects are instantiated.

6-5-17 Object Instantiation:

Like: `A a(2);` `A* a=new A(3);`

When we have a variable declaration as the first one above we do this:

- subject initialization was done first, when the function block or program block was encountered, so the variable (a) has already a place reserved for it and enough for all it's member variables
- so the step we need to make is calling the constructor with the passed arguments
- push parameters on the stack (there is no return value here)
- make a function call command for that constructor
- pop all stacked parameters.

So this will do the work and our variable gets initialized.

For the second case whether it was at variables declaration or any where else:

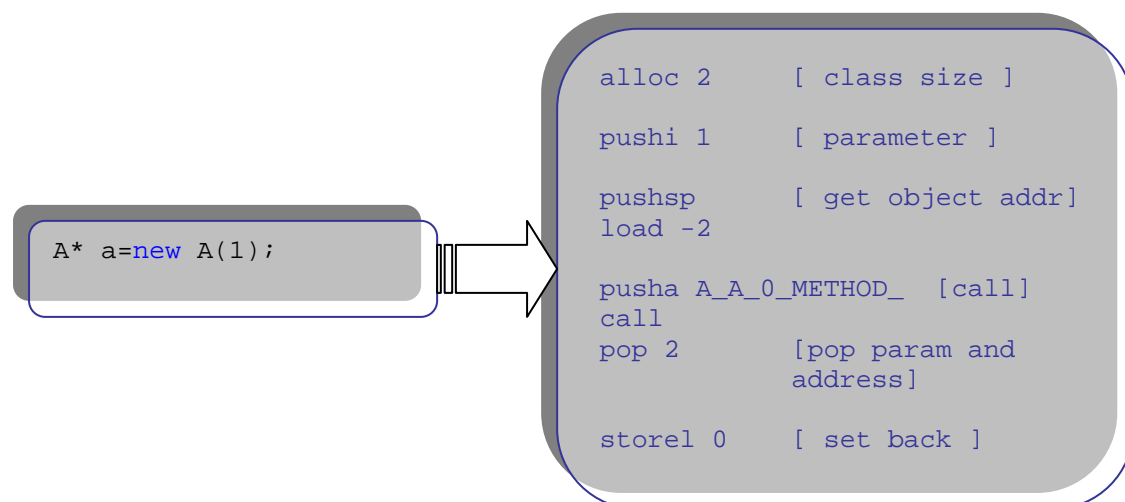
`a=new A(3);`

in this case the object is allocated on the fly on top of the stack, then it's set back into the variable, steps are:

- allocate new block of the class size
- generate the constructor parameters and push them into the stack
- push stack pointer onto the stack
- load the address `-params_call_count-1` which in this case will be the address allocated earlier and represents the object's address
- now push the constructor call
- pop `params_call_count+1`
- now you can set variable back

the only special thing about the new object is dealing with stack pointer, which is the only place we need it.

For example:



6-5-18 Generating math operation code:

For the assignment case, or normal math operations it's simply making the right typed command, for example:

The most common operand is integer for summing operation then issue ADD

The most common operand is double for a multiplication operation the issue FMUL

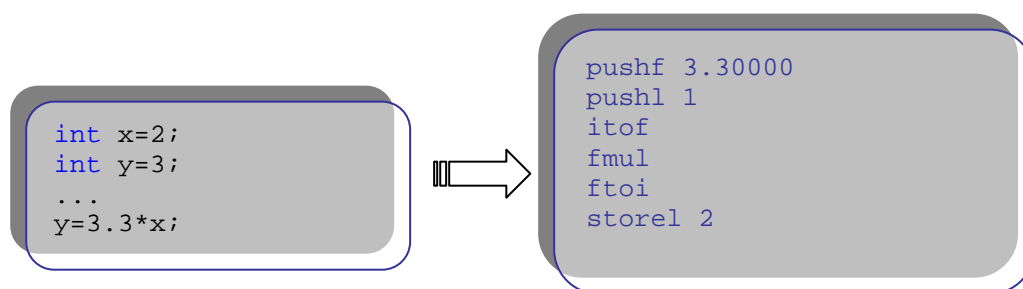
And so on.

6-5-19 Generating Cast Balancing Code:

As we saw in the assignment case, we need to make the proper cast operation if operand types mismatches.

Type checking allows this when explicit or implicit types conversion is allowed.

We represent int, char, bool as integers, so the only cast that might happen is when one operand is double and the other is something else, so we issue `itof` command.



The example above shows a sample of math operation and implicit type conversion, which up in we converted the integer value x into float then multiplied and converted the whole back into integer.

This is a very important feature, since this way we don't lose the precision, we always convert to the most common type.

6-5-20 unary operations:

All unary operations are direct use of the operations available by the vm, but for the increment operation:

++ expr; here we increment then push the expression on the stack
 Expr++ we push the expression on the stack then issue the increment
 operation
 the same apply for decrement
 not negate with not command

6-5-21 logical operation code:

```

expr > expr
expr < expr
expr >= expr
expr <= expr
expr == expr
expr != expr
expr && expr
expr || expr

```

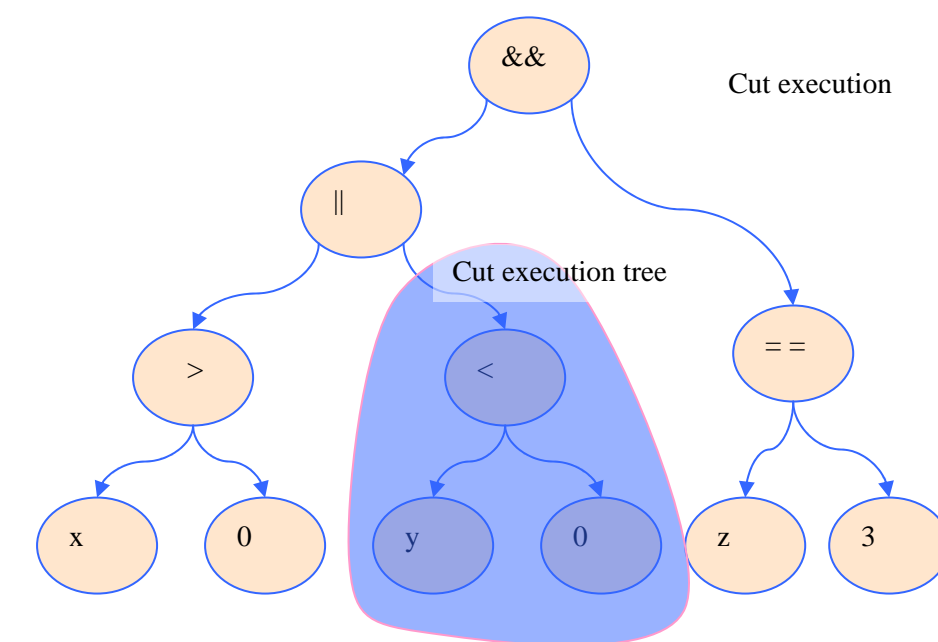
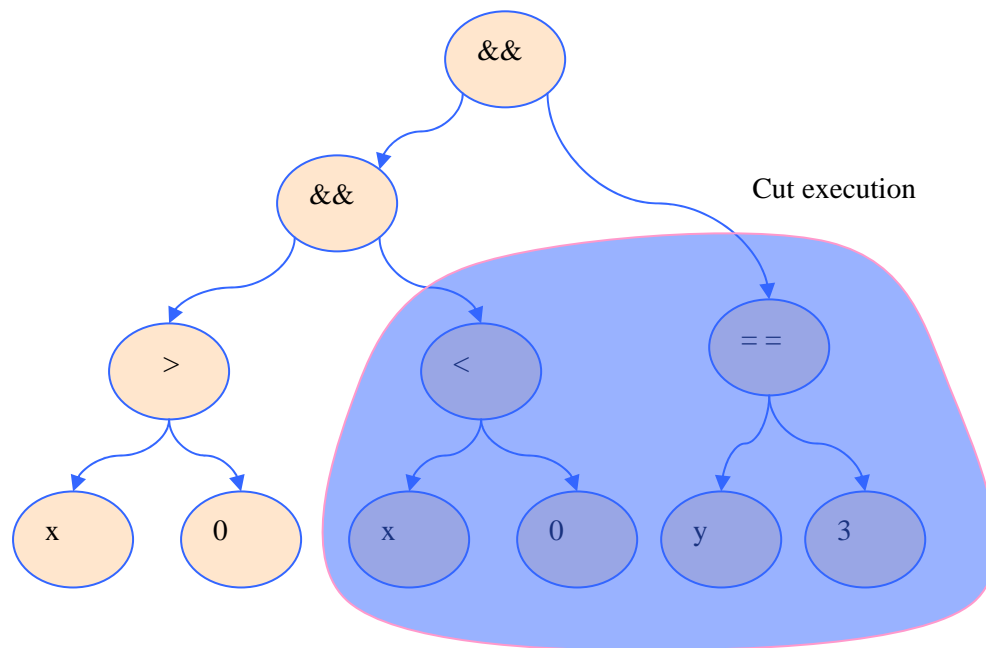
the general approach for this:

- generate expression code for the right operand
- generate cast code for it, to the most common node
- generate left operand expression code
- generate left operand cast to the most common node
- issue the proper command (INF FINF, SUP FSUP, INFEQ, FINFEQ, FUPEQ, SUPEQ, EQUAL, NOT...)

of course expression casting may not output any statements if no need for that.

The special part about the logical operations is the AND-OR operations, which we talked about in code optimization, we'll talk about it back here.

$(x > 0) \&\& (x < 0) \&\& (y == 3)$



$((x > 0) \parallel (y < 0)) \&\& (z == 3)$

The AND-OR tree cut

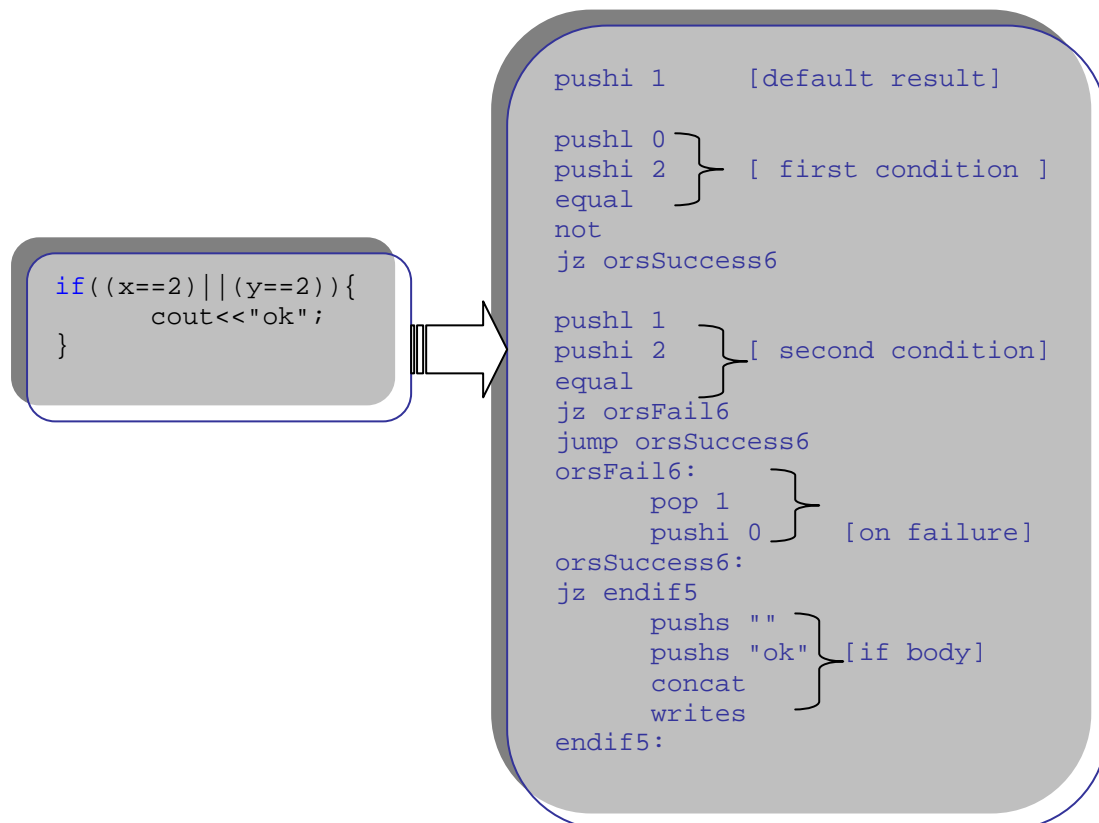
We implement this mechanism by using labeling and jump statements:

6-5-22 The AND-OR tree cut:

We push 1 then try the tree if failed we pop it back and push 0 instead

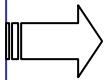
- generate or success label, and or fail label
- push 1 first as success by default unless opposite is proven
- generate node1 expression
- negate the expression with not command
- on 0 jump to or success label
- generate the second operand expression
- on 0 jump to ors failure label
- else jump to ors success label
- output ors fail label
- pop 1 (the suggested success code)
- push 0 back (failure was proven)
- output ors success code

it might like a little bit awkward, but it has the best results, and the best performance, we tested it under the Graphical VM and it did what it's supposed to do



To the right is a sample code generated for the left example, which is an or tree sample with two operands.

```
if((x==2)&&(y==3) || (x>0)){
    cout<<"ok";
}
```



```
pushi 1      [init And]

pushl 0
pushi 2      [first operand of and]
equal
jz andsFail6

pushi 1      [init or]
pushl 1
pushi 3      [first or operand]
equal
not
jz orsSuccess8

pushl 0
pushi 0      [second or operand]
SUP
jz orsFail8

jump orsSuccess8
orsFail8:
pop 1        [or fail action]
pushi 0

orsSuccess8:

jz andsFail6
jump andsSuccess6
andsFail6:
pop 1        [and fail action]
pushi 0
andsSuccess6:
jz endif5
    pushes ""
    pushes "ok" [ if body ]
    concat
    writes
endif5:
```

This AND-OR schema works ever where and in any depth complexity.
For the and tree case, we don't negate the first operand (mostly it depends on the default pushed value 1 and the or fail action)

6-5-23 cout statement code:

```
Cout<<x<<"test"<<y*z<<u+2+3<<"\n"<<x*x;
```

vm issues one write command and new line will be entered, so we needed to this:

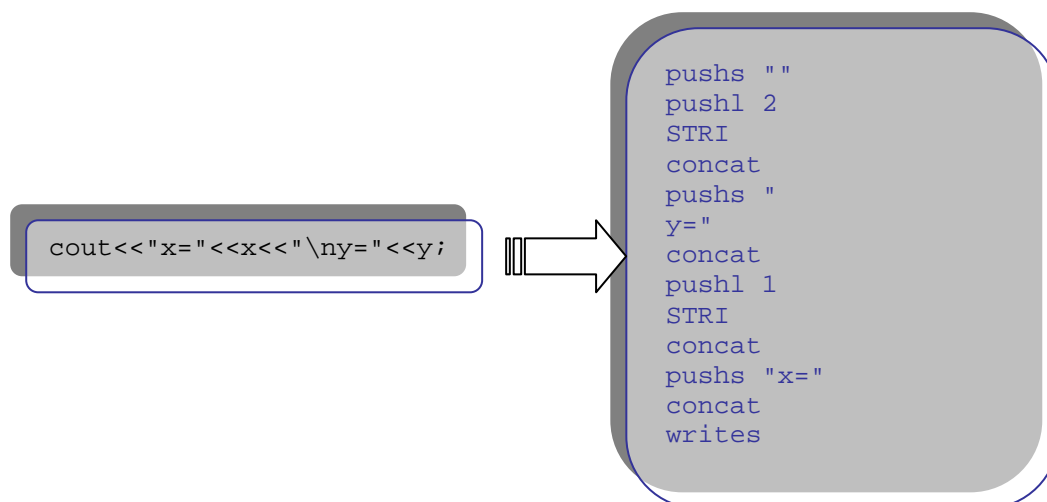
we concatenate the input list into one string

- push initial empty string
- go through out put list and issue the proper convert command, we convert all types into string using STRI, STRF
- concatenate this converted string with the previous one (after flipping the order)
- finally issue writes command.

For constants of type string which are becoming char*, we tried to issue a direct writes for them but the special characters were printed normally, so we needed to modify these strings to set each special character with it's ASCII value, so, we iterate through every character and matched the current character with the previous ones to find a match:

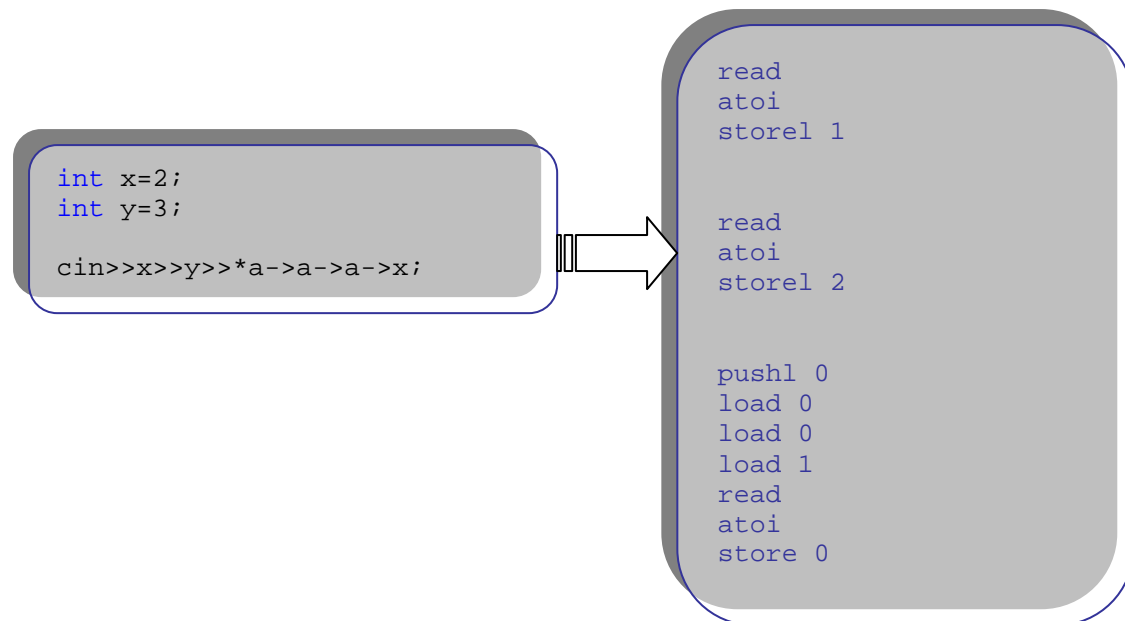
- [\n] sequence [two characters] is replaced by [10 32] which is the ASCII space and new line characters
- [\t] sequence of two characters is replaced by [9 32] which is the ASCII [space tab] characters
- [\"] replaced by [' space]

By doing this operation the strings could outputted well formatted.

**6-5-34 cin statement code:**

```
Cin>>x>>y>>z;
```

The input list is broken down into it's single elements, then read



The elements are read the same order they are entered, the final element shows reading a complex variable which was mentioned earlier.

Info is always read as strings, so we need to convert them to the proper types.

This is all about code generation, next we'll present some real world examples and their equivalent code.

7- Real World Examples:

7-1 Example 1 source Code: Factorial

```

/*
testing recursive functions:
factorial examples:
*/
int fact(int x){
    if(x<=1)
        return 1;
    return x*fact(x-1);
}

int main(){
    int x=0;
    int choice=1;
    cout<<"testing factorial...";
    while(choice!=0){
        cout<<"input a number to factorize";
        cin>>x;
        cout<<"factorial of x="<<fact(x);
        cout<<"*****\ninput a choice: 0 to exit";
        cin>>choice;
    }
    return 0;
}

```

Example 1 VM Code: Factorial

```

alloc 0

start
    pushi 0
    pusha main_METHOD_
    call
    pushs "main exit with
code:"
    writes
    writei
stop
fact_METHOD_:
    pushi 0
    pushl -1
    storel 0
    pushl 0
    pushi 1

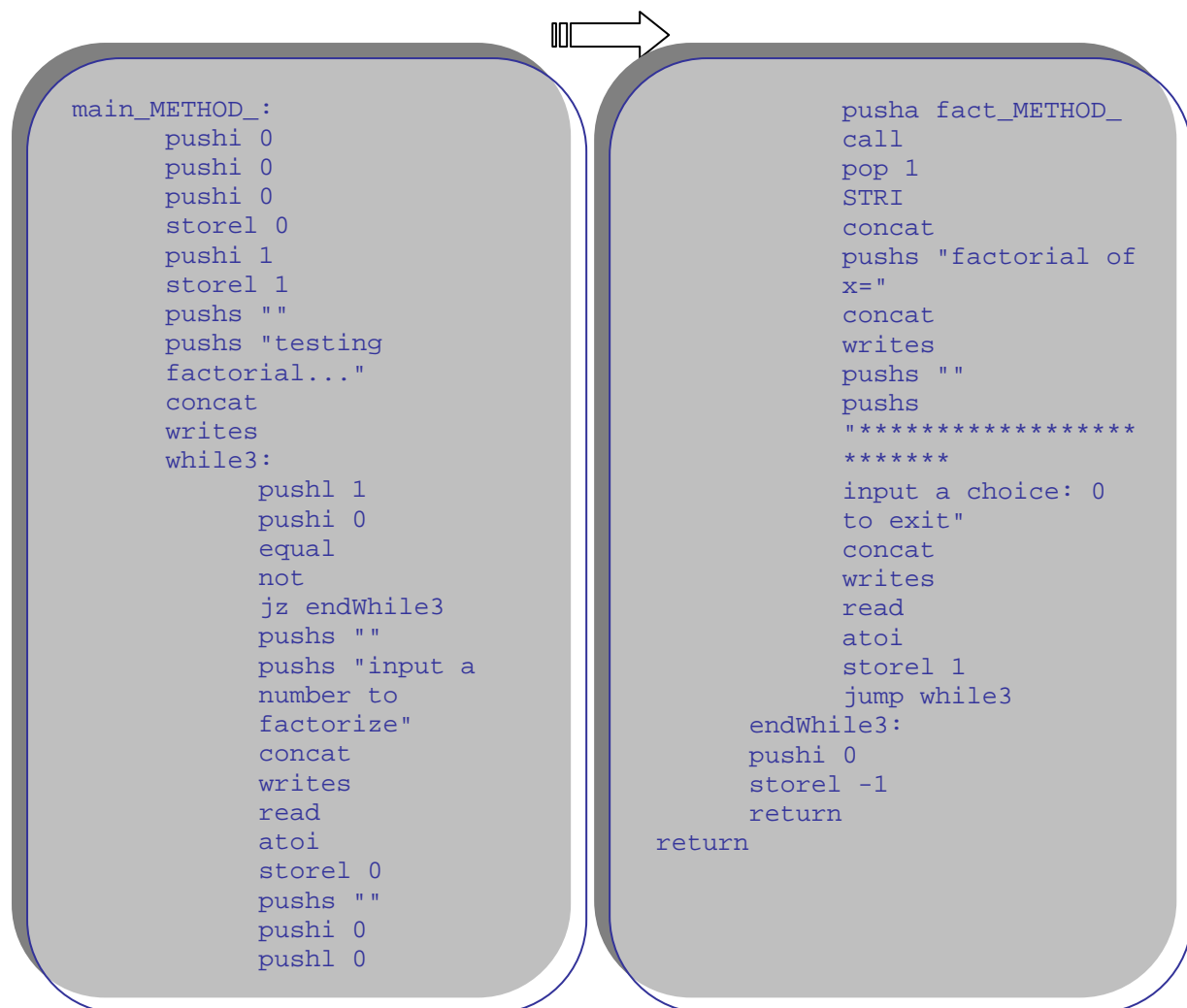
```



```

        jz endif1
        pushi 1
        storel -2
        return
endif1:
    pushl 0
    pushi 0
    pushl 0
    pushi 1
    sub
    pusha fact_METHOD_
    call
    pop 1
    mul
    storel -2
    return
return

```



Example 1 execution: Factorial

```

C:\ E:\WINXP\System32\cmd.exe
E:\MiniC++2>vm 1.vm
VM version 1.0
testing factorial...
input a number to factorize
read =>
2
factorial of x=2
*****
input a choice: 0 to exit
read =>
1
input a number to factorize
read =>
5
factorial of x=120
*****
input a choice: 0 to exit
read =>
0
main exit with code:
0
E:\MiniC++2>_
  
```

This program continues asking for a choice until 0 is put, the number to factorize is out put
 This program shows sample loop and sample output concatenation, and function calls

The program exit with code 0

7-2 Example 2 source code: Lined List

```
/*
simple linked list application
here we use the class like we use
the struct just to hold information nothing more
*/

class Record{
    Record(int);
    int val;
    Record* next;
};

Record::Record(int val){
    this->val=val;
    next=NULL;
}

int main()
{
    Record* head=new Record(0);
    Record* temp=head;

    //linked list creation
    for(int i=0;i<10;i++){
        temp->next=new Record(i);
        temp=temp->next;
    }

    cout<<"linked list was created...";
    /*linked list testing using the for loop
    this loop down here, will break when temp
    becomes NULL, which in our case will go 10 iterations
    only if the list didn't terminate a runtime error will
    be issued by vm
    */
    cout<<"testing for loop over the linked list";
    temp=head;
    for(int j=0;j<15;j++){
        cout<<temp->val;
        temp=temp->next;
        if(temp==NULL)
            break;
    }

    temp=head;
    cout<<"testing while loop over the linked list";
    //linked list test using the while loop
    while(temp!=NULL){
        cout<<temp->val;
        temp=temp->next;
    }

    return 0;
}
```

Example 2 VM Code: Linked List

```

alloc 0

start
    pushi 0
    pusha main_METHOD_
    call
    pushes "main exit with code:"
    writes
    writei
stop

Record_Record_METHOD_:
    pushi 0
    pushl -2
    storel 0
    pushl -1
    pushl 0
    store 1
    pushl -1
    pushg 0
    store 0
return

main_METHOD_:
    alloc 0
    alloc 0
    pushi 0
    pushi 0
    alloc 2
    pushi 0
    pushsp
    load -2
    pusha Record_Record_METHOD_
    call
    pop 2
    storel 0
    pushl 0
    storel 1
    pushi 0
    storel 2
    forl:
        pushl 2
        pushi 10
        INF
        jz endFor1
        pushl 2
        dup 1
        pushi 1
        add
        storel 2
        pushl 1
        alloc 2
        pushl 2
        pushsp
        load -2
        pusha
        Record_Record_METHOD_
        call
        pop 2
        store 0
        pushl 1

```



```

load 0
storel 1
jump for1

endFor1:
    pushes ""
    pushes "linked list was created..."
    concat
    writes
    pushes ""
    pushes "testing for loop over the
    linked list"
    concat
    writes
    pushl 0
    storel 1
    pushi 0
    storel 3

    for3:
        pushl 3
        pushi 15
        INF
        jz endFor3
        pushl 3
        dup 1
        pushi 1
        add
        storel 3
        pushes ""
        pushl 1
        load 1
        STRI
        concat
        writes
        pushl 1
        load 0
        storel 1

        pushl 1
        pushg 0
        equal
        jz endif5
        jump endFor3
    endif5:
        jump for3
endFor3:

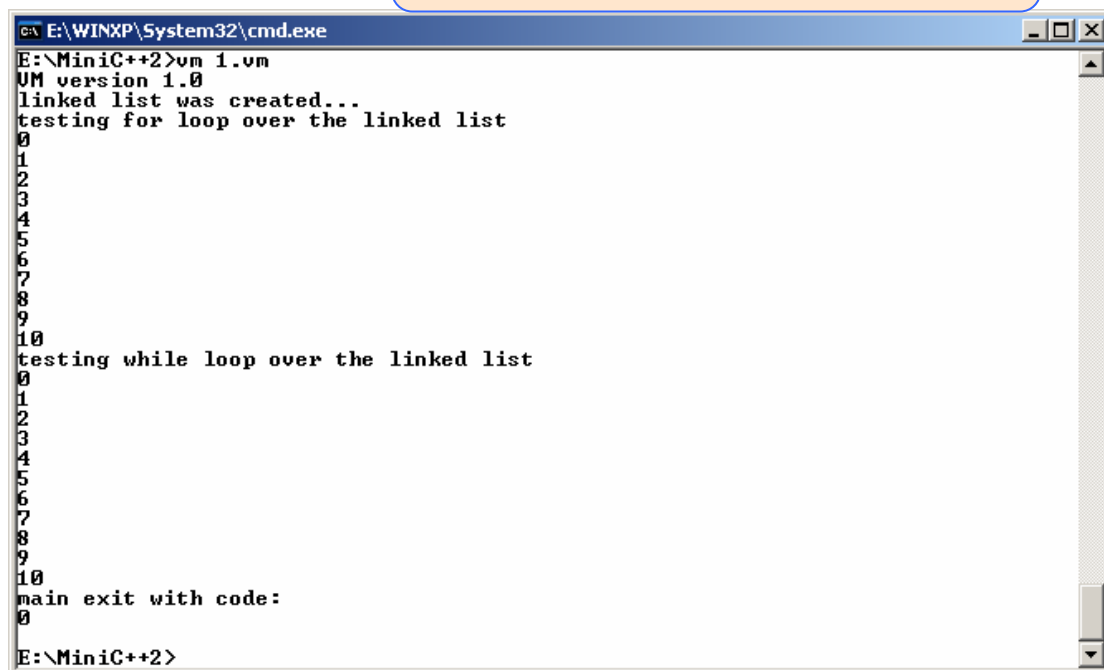
    pushl 0
    storel 1
    pushes ""
    pushes "testing while loop over the
    linked list"
    concat
    writes

    while7:
        pushl 1
        pushg 0
        equal
        not
        jz endWhile7

```

```
        pushs " "  
        pushl 1  
        load 1  
        STRI  
        concat  
        writes  
        pushl 1  
        load 0  
        storel 1  
        jump while7  
    endwhile7:  
    pushi 0  
    storel -1  
    return  
return
```

Example 2 VM alive: Linked List



```
E:\WINXP\System32\cmd.exe  
E:\MiniC++2>vm 1.vm  
VM version 1.0  
linked list was created...  
testing for loop over the linked list  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
testing while loop over the linked list  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
main exit with code:  
0  
E:\MiniC++2>
```

The above example show some basic operations and tests, like for, while, if, break, function call, pointers, local variables, runtime allocation with new, classes, constructors, output statements...

The program returns the code 0 after it exited.

7-3 Example 3 source code: Multi level inheritance

```

class A{
public:
    A();
    int a;
};

A::A(){
    cout<<"A constructor called...\n";
    a=0;
}

class B: public A{
public:
    B();
    B(int);
    int b;
};

B::B(){
    cout<<"B first constructor called...";
    b=0;
}

B::B(int x){
    b=x*2;
    cout<<"B second Constructor called...\n";
}

class C: public B{
public:
    C(int x);
    int c;
};

C::C(int x):B(x){
    c=x;
    cout<<"C was constructed...";
}

int main(){
    C c(2);
    cout<<"  c.c="<<c.c<<"  c.b="<<c.b<<"  c.a="<<c.a;
    return 0;
}

```

The steps for initialization when constructor called is like this:

- check if current class constructor definition has any base initialization list or not.
- if it has then, check if it's element was a class init list
- call the base list appropriate constructor

If the base list exists but the initialization doesn't specify the base constructor then we call the default constructor which must exist in this case, otherwise a type check error is issued.

Above is a three level inheritance example (any level allowed), the C constructor can explicitly specify the base constructor to be called, and thus here we call the one with the integer parameter, which gets doubled in the base, the base B has no explicit constructor and thus the default parent constructor must be provided and thus called.

And the generated code for this:

Example 3 VM code: Muli level inheritance

```

alloc 0
start
pushi 0
pusha _function_main_66
call
pushs "main exit with code:"
writes
writei
stop
_CLASS_A_A_7:
pushs ""
pushs "A constructor
called..."
"
concat
writes
pushl -1
pushi 0
store 0
return
_CLASS_B_B_19:
pushl -1
pusha _CLASS_A_A_7
call
pop 1
pushs ""
pushs "B first constructor
called..."
concat
writes
pushl -1
pushi 0
store 1
return
_CLASS_B_B_26:
pushi 0
pushl -2
storel 0
pushl -1
pusha _CLASS_A_A_7
call
pop 1
pushl -1
pushl 0
pushi 2
mul
store 1
pushs ""
pushs "B second Constructor
called..."
"

```

```

concat
writes
return

_CLASS_C_C_41:
pushi 0
pushl -2
storel 0
pushl 0
pushl -1
pusha _CLASS_B_B_26
call
pop 2
pushl -1
pushl 0
store 2
pushs ""
pushs "C was constructed..."
concat
writes
return
_function_main_66:
alloc 3
pushi 2
pushl 0
pusha _CLASS_C_C_41
call
pop 2
pushs ""
pushl 0
load 0
STRI
concat
pushs " c.a="
concat
pushl 0
load 1
STRI
concat
pushs " c.b="
concat
pushl 0
load 2
STRI
concat
pushs " c.c="
concat
writes
pushi 1
storel -1
return
return

```


The execution was like:

Example 3 alive: Multi-level inheritance



```
c:\D:\MAINXP\System32\cmd.exe
E:\MiniC++2>vm 1.vm
VM version 1.0
A constructor called...
B second Constructor called...
C was constructed...
  c.c=2  c.b=4  c.a=0
main exit with code:
1
E:\MiniC++2>_
```

It shows the construction sequence (no destruction was called, though destructors were implemented like constructors, but we didn't suggest methods for calling them) (this example contains different naming, due naming changes which we didn't changed the rest of the report examples for).

Now we present our final piece in this thesis, which contains the most important features of our compilers: **Object Oriented Stack Application:**

**7-4 Example4 source code:
Stack application**

```
/**stack class and test application
*/

class Record{
public:
    Record(int);
    int val;
    Record* next;
};
Record::Record(int val){
    this->val=val;
    next=NULL;
}

class Stack{
private:
    Record* head;
public:
    Stack();
    Record* push(Record* rec);
    Record* pop();
    void print();
};

Stack::Stack(){
    cout<<"stack created";
    head=NULL;
}
Record* Stack::push(Record* rec){
    rec->next=head;
    head=rec;
    return head;
}
Record* Stack::pop(){
    Record* temp=head;
    if(head!=NULL)
        head=head->next;
    return head;
}
void Stack::print(){
    Record* temp=head;
    while(temp!=NULL){
        cout<<temp->val;
        temp=temp->next;
    }
}

int main(){
    Stack* stack=new Stack();
    for(int i=0;i<10;++i){
        stack.push(new Record(i));
    }
    cout<<"stack before popping:";
    stack->print();
    cout<<"\nstack after popping:";
    stack.pop();
    stack.pop();
    stack.print();
}
```

The previous code is a functional one, but before that some errors were happening and error messages were issued like

```

e:\MiniC++2\Debug\minic++.exe
error: [154,6] missing < :
error: [127,2] can't assign, type mismatch
error: [131,12] variable not found
error: [131,2] can't assign, type mismatch
error: [132,2] variable not found
error: [132,2] can't assign, type mismatch
error: [133,9] variable not found
error: [133,2] return type mismatches definition
can't generate code for incorrect code
*****symbol table content*****
rec_address  parent_address  depth  type  name
0x0050B780    0x00321228      1      113   push
0x0050ACE8    0x0050A3A0      2      113   push
0x00321228    0x00000000      0       0   _File_Scope_Rec_
0x0050A0B8    0x005098C0      3      111   val
0x0050D940    0x0050D0A0      2      111   i
0x0050C908    0x0050B2D0      3      111   temp
0x0050C248    0x0050B0C0      3      111   temp

```

Sample error reporting.

Now after correction, the code generated was:

Example4 VM code: Stack application

```

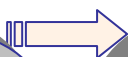
alloc 0
start
pushi 0
pusha _function_main_103
call
pushs "main exit with code:"
writes
writei
stop
_CLASS_Record_Record_9:
pushi 0
pushl -2
storel 0
pushl -1
pushl 0
store 1
pushl -1
pushg 0
store 0
return
_CLASS_Stack_Stack_29:
pushs ""
pushs "stack created"
concat
writes
pushl -1
pushg 0
store 0
return

```

```

_CLASS_Stack_push_47:
alloc 0
pushl -2
storel 0
pushl 0
pushl -1
load 0
store 0
pushl -1
pushl 0
store 0
pushl -1
load 0
storel -3
return
return
_CLASS_Stack_pop_64:
alloc 0
pushl -1
load 0
storel 0
pushl -1
load 0
pushg 0
equal
not
jz endif1
pushl -1
pushl -1
load 0

```

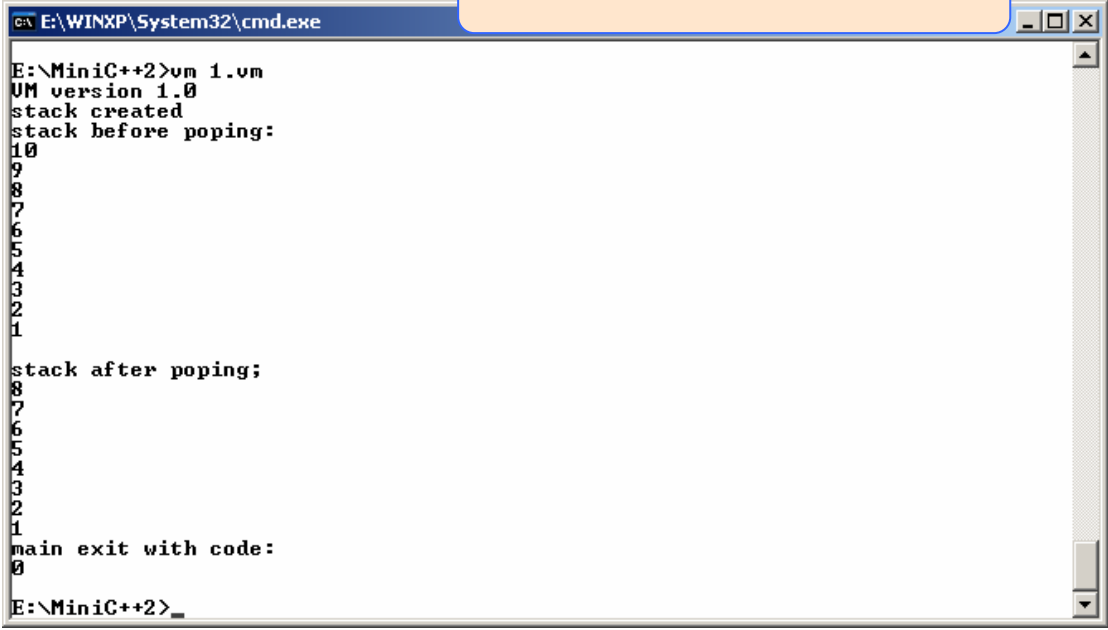


```
load 0
store 0
endif1:
pushl -1
load 0
storel -2
return
return
_CLASS_Stack_print_81:
alloc 0
pushl -1
load 0
storel 0
while3:
pushl 0
pushg 0
equal
not
jz endWhile3
pushs ""
pushl 0
load 1
STRI
concat
writes
pushl 0
load 0
storel 0
jump while3
endWhile3:
return
_function_main_103:
alloc 0
pushi 0
alloc 1
pushsp
load -1
pusha _CLASS_Stack_Stack_29
call
pop 1
storel 0
pushi 0
storel 1
for5:
pushl 1
pushi 10
INF
jz endFor5
pushl 1
pushi 1
add
dup 1
storel 1
alloc 0
alloc 2
pushl 1
pushsp
load -2
pusha CLASS Record Record 9
```

```
call
pop 2
pushl 0
pusha _CLASS_Stack_push_47
call
pop 3
pop 1
jump for5
endFor5:
pushs ""
pushs "stack before popping:"
concat
writes
pushl 0
pusha _CLASS_Stack_print_81
call
pop 1
pushs ""
pushs "
stack after popping;"
concat
writes
alloc 0
pushl 0
pusha _CLASS_Stack_pop_64
call
pop 1
alloc 0
pushl 0
pusha _CLASS_Stack_pop_64
call
pop 1
pushl 0
pusha _CLASS_Stack_print_81
call
pop 1
return
```

And the execution of this applications:

Example4 alive: Stack application



```
E:\WINXP\System32\cmd.exe
E:\MiniC++>vm 1.0m
VM version 1.0
stack created
stack before popping:
10
9
8
7
6
5
4
3
2
1
stack after popping:
8
7
6
5
4
3
2
1
main exit with code:
0
E:\MiniC++>_
```

There are plenty of tests to provide, but the ones before are the ones that matters the most, they show the capacity of this compiler to do the basic things in a programming language.

We apologize if any words were mistyped, but time is a killer...

That's all we got.

The final page is a sample class diagram of our project, though developed using C++ how ever it's not that object oriented application.

The middle class (the giant one) is a sample for why we need not push all other methods into the view, so one sample class with it's methods is presented.

The End.

Supplement

