How to Construct Declarations and New .dep Files

This document describes how to deal with declarations of objects of various types and how to creaty a new empty .dep file. The source code can be found in the sage/Sage++/demos/new_file directory.

1 Working with types: building variable and parameter declarations for C and C++ programs

To create a new variable and declare it in a given scope requires three steps. First, a type chain must be created. Second, a symbol of that type is created in the given scope. Third, a declaration statement is generated.

In C and C++ variable declarations can be very complicated, because the full type is really described in the C source text as a combination of type attributes and a reference expression. For example,

```
int & (*x)[10];
```

has part of the declaration that consists of the type name "int" and the rest is given by the expression

```
 & (*x)[10]
```

Sage++ has another way to describe the type of this object that is closer to how the programmer views it. In this case, the type of x is

a pointer to
an array of
references to
integers.

This is represented in Sage as a chain of type objects of the form:

SgPointerType → SgArrayType → SgReferenceType → SgTypeInt.

The reference expression, & (*x)[10], in the declaration is represented by Sage++ expressions as a chain of the form

SgAddrOp → SgPntrArrRefExp → SgDerefOp

The actual dimension list [10] is an expression list of integer values shared by both the SgPntrArrRefExp and the SgArrayType objects.

Note that the reference expression is related to the reverse of the type chain, so it is possible to build the reference expression chain given the type chain.

1.1 The SgMakeDeclExp function and related methods in SgSymbol class

The function that will automatically create the correct expression chain from the type chain is

```
SgExpression * SgMakeDeclExp(SgSymbol *x, SgType *type_chain);
```

This is a new function, that has been added in Sage++ version 1.7. There is also a method in the SgSymbol class:

```
SgExpression *SgSymbol::makeDeclExpr()
```

(note the subtle differences in spelling!) that does the same thing; it actually calls the SgMakeDeclExp function; the type is set to the type associated with the symbol of interest.

While the user may need to call the SgMakeDeclExp function in some situations, there are more convenient ways to create a declaration without explicitly doing this.

For example, let "fun" be a pointer to s SgFunctHedrStmt object created as in

```
SgFuncHedrStmt *fun = new SgFuncHedrStmt("foo");
```

Then, then to add the local variable declaration

```
int *z; // - a pointer to an int.
```

we

1. create the type chain,
2. create the symbol and
3. declare it.

```
SgType *p = new SgPointerType(*SgTypeInt());
SgSymbol *sym = new SgVariableSymb("z", *p, *fun);
sym->declareTheSymbol(*fun);
```

The method

```
void SgSymbol::declareTheSymbol(SgStatement &st)
```

also calls the SgMakeDeclExp function; but, in addition, it takes care of inserting the declaration in a proper position determined by the scope specified by the SgStatement &st argument.

A more interesting example is the pointer to array of character references described above.

```
char & (*y)[3]; // - a pointer to an array of char refs.
```

The three steps are shown below. Notice that the type chain is created from the end of the chain to the front.

```
SgType *q = new SgReferenceType(*SgTypeChar());
SgArrayType *ar = new SgArrayType(*q);
ar->addDimension(&SgValueExp(3));
q = new SgPointerType(*ar);  // q is the final type chain.
sym = new SgVariableSymb("y", *q, *fun);
sym->declareTheSymbol(*fun);
```

There are some cases where declareTheSymbol() method is not the best approach. For example, it will not work for a declaration with an initializer, as in:

```
int w[] = {1,2};  // an initialized array.
```

It is easiest to create the variable declaration statement (SgVarDeclStmt) explicitly and then add the expression for the initializer. The code for this example is shown below.

```
// create the int w[]; declaration statement.
SgArrayType *ar = new SgArrayType(*SgTypeInt());
ar->addDimension(NULL);
SgSymbol *sym = new SgVariableSymb("w", *ar, *fun);
SgVarDeclStmt *decl = sym->makeVarDeclStmt();

// create the initializer list. {1, 2}
SgExpression * one = new SgValueExp(1);
SgExpression * two = new SgValueExp(2);
SgExprListExp * ls = new SgExprListExp(*one);
ls->append(*two);
SgExpression * in_ls = new SgInitListExp(*ls);
decl->setInitialValue(0, *in_ls);

// explicitly add the declaration to the function body.
fun->insertStmtAfter(*decl);
```

The method

```
SgVarDeclStmt *SgSymbol::makeVarDeclStmt()
```

is related to the other methods described above (it also calls the SgMakeDeclExp function).

Another important case involves adding qualifiers (like const, long, short, etc.) to variables. For example, consider the following

```
const int x;
```

The const qualifier must be added as a modifier of the type of int. This is accomplished by using a SgDescriptType object which has a special bit field for describing the modifier. The proper type chain is given by

```
p = new SgDescriptType(*SgTypeInt(), BIT_CONST);
sym = new SgVariableSymb("x", *p,  *fun);
sym->declareTheSymbol(*fun);
```

Unfortunately, this is not completely general. In the declaration

```
int * const x;
```

the pointer type is constant but not the integer. We do not use a SgDescriptType node for this task. Instead, we set the modifier flag in the SgPointerType object as follows.

```
SgPointerType *z = new SgPointerType(*SgTypeInt());
z->setModifierFlag(BIT_CONST);
sym = new SgVariableSymb("x", *z,  *fun);
sym->declareTheSymbol(*fun);
```

1.2 Function Prototypes

Function prototypes are another type of variable declaration. The are a bit harder to construct than ordinary variables. For example, to build the prototype

```
char f(int);
```

we first must create a function name symbol whose type is the type of the returned value of the function. Function symbols are referred to in expressions by SgFunctionRefExp in much the same way that regular variables are referred to by SgVarRefExp variable reference objects. A SgFunctionRefExp has a pointer to the symbol object for the function and a list of arguments. The

```
SgExpression * SgFunctionRefExp::AddArg( char *name, SgType &t)
```

method takes as arguments the name of the argument (which can be an empty string for prototypes) and the type. AddArg automatically creates the reference expression for the argument. Once the SgFunctionRefExp expression is generated, then it must be added to an expression list and a declaration statement must be created as shown below.

```
SgFunctionSymb *proto = new SgFunctionSymb(FUNCTION_NAME, "f", *SgTypeChar(),
               *fun);
SgFunctionRefExp *fref = new SgFunctionRefExp(*proto);
fref->AddArg("", *SgTypeInt());
SgExpression *ls = new SgExprListExp(*fref);
SgStatement *prot_st = new SgVarDeclStmt(*ls, *SgTypeChar());
fun->insertStmtAfter(*prot_st);
```

1.3 Adding parameters to function headers

When building a new function, one must often add parameters. The AddArg method described above is used to add formal parameters to function reference expressions. There is another AddArg method:

```
SgExpression * SgProcHedrStmt::AddArg(char *name, SgType &t)
```

that is used to add formal parameters to function header statements (the class SgFuncHedrStmt is a subclass of SgProcHedrStmt). For example, given a function "fun" constructed as above with

```
SgFuncHedrStmt *fun = new SgFuncHedrStmt("foo");
```

we can add formal parameters to this by using AddArg. For example, to create

```
int foo( char * x[][10]){
}
```

we proceed exactly as with a variable declaration: build the type chain for the parameter and add the parameter with AddArg. In this case the type chain is

SgArrayType → SgPointerType → SgTypeChar

and the code to construct this object and add it as a parameter is given by the following.

```
  SgType *q = new SgPointerType(*SgTypeChar());
  SgArrayType *ar = new SgArrayType(*q);
  ar->addDimension(NULL);
  ar->addDimension(&SgValueExp(10));
  fun->AddArg("x",*ar);
```

Note that AddArg automatically creates the symbol object for the argument and places it in the correct scope.

1.4 Pointers to Functions

To create a variable or formal parameter that is a pointer to a function we use the SgFunctionType class. The parameter

```
int & (*f)()
```

is a pointer to a function that returns a reference to an integer and has type chain

SgPointer → SgFunction → SgReference → SgTypeInt

The code to generate this object and add it as a parameter to our function foo is given below.

```
SgType* q = new SgReferenceType(*SgTypeInt());
q = new SgFunctionType(*q);
q = new SgPointerType(*q);
fun->AddArg("f",*q);
```

Finally, consider the case of a pointer to a function with a more complete prototype, i.e. one that lists the arguments to functions, such as

```
int & (*f)( int &, char *)
```

This object is a pointer to a function that returns a reference to an int and takes a reference to an int and a pointer to a char as parameters. To build this object, we first create the pointer-to-function object and add it as a parameter to our base function "fun". Then we use the fact that the AddArg() function returns the declaration expression. With the declaration expression, we can find the SgFunctionPntrExp node and add the remaining parameters to that object. The AddArg method used here is:

```
SgExpression * SgFuncPntrExp::AddArg(SgSymbol *f,  char *name, SgType &t).
```

It requires an additional parameter.

```
SgType * q = new SgReferenceType(*SgTypeInt());
SgType * p = new SgFunctionType(*q);
q = new SgPointerType(*p);
SgExpression *exp = fun->AddArg("f",*q);

// find the function_op (SgFuncPntrExp) node.
while(exp && (exp->variant() != FUNCTION_OP)) exp = exp->lhs();
SgFuncPntrExp *fptr = isSgFuncPntrExp(exp);

// find the symbol generated by AddArg
e = exp;
while(e && !isSgArrayRefExp(e) && !isSgVarRefExp(e)) e = e->lhs();
SgSymbol *fname = e->symbol();

// now add the arguments to f.
q = new SgReferenceType(*SgTypeInt());
fptr->AddArg(fname, "", *q);
q = new SgPointerType(*SgTypeChar());
fptr->AddArg(fname, "", *q);
```

2 Creating a new empty .dep file

Usually, the .dep file is created by a parser and a Sage program modifies an already existing .dep file. However, in some situations, there is a need for creating an empty .dep file and then adding statements to it. In Sage++ version 1.7, a new constructor:

```
SgFile::SgFile(int Language, char * dep_file_name)
```

has been added. It can be used to create new, empty .dep files. The code fragment below shows an example of usage:

```
SgProject project("x.proj");
SgFile file(CSrc, "y.c");  // create a new file
SgStatement *s = f->firstStatement();
SgPointerType *p, *q;
SgType *v;
if(s == NULL) printf("no first statement\n");
SgFuncHedrStmt *mane = new SgFuncHedrStmt("main");
mane->AddArg("argc",*SgTypeInt());  // add int argc
v = new SgPointerType(*SgTypeChar());
q = new SgPointerType(*v);
// note: new SgPointerType (*p) will not work,
//  the reason is that the default copy constructor is called
mane->AddArg("argv", *q); // char ** argv
s->insertStmtAfter(*mane);
```

Note that the "empty" file created by the SgFile constructor is not really completely empty: it contains one statement (the global node). Also, in order for the above example to work, you need to have x.proj and x.dep files in the same directory.