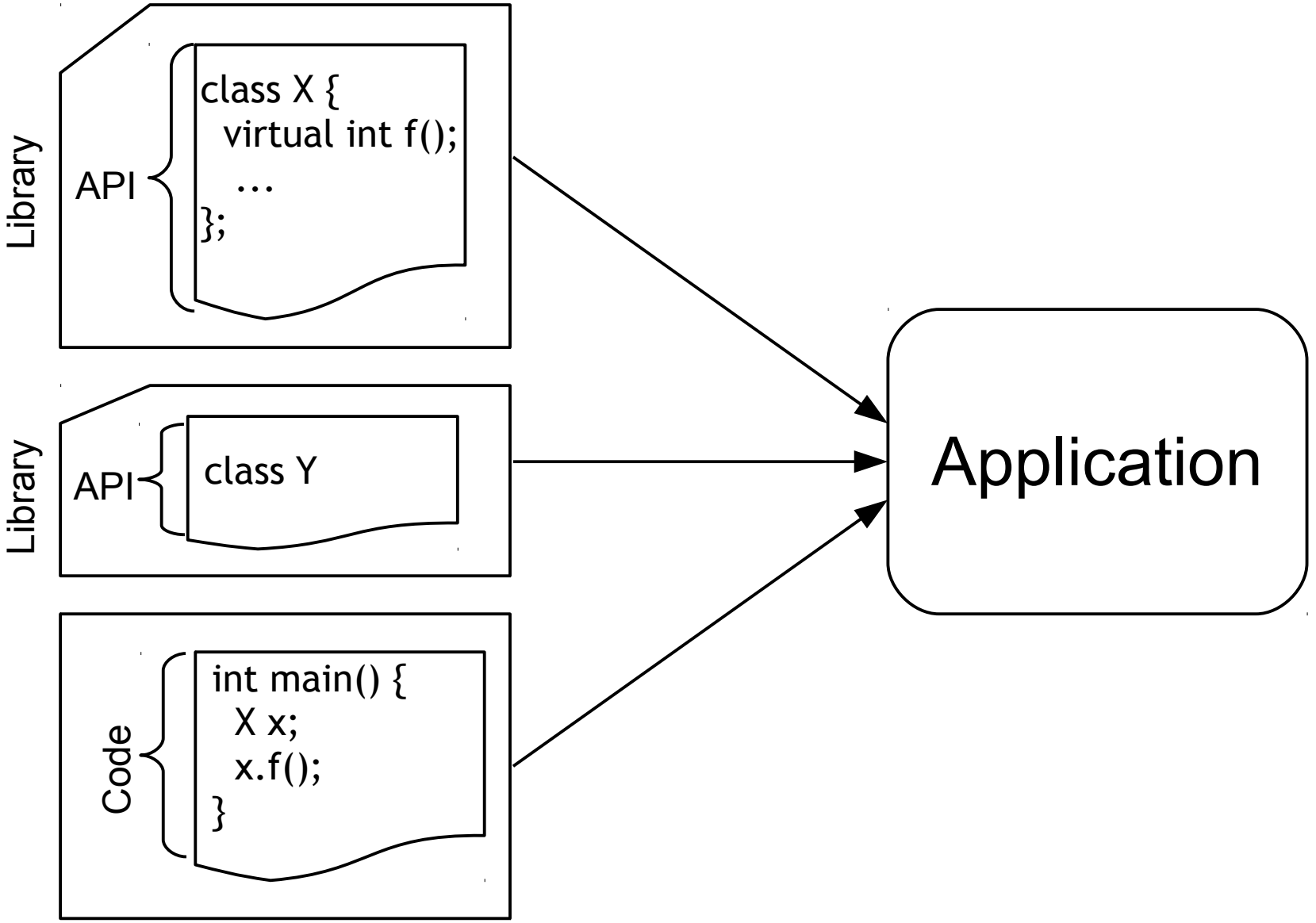
# ABI Compatibility Through A Customizable Language

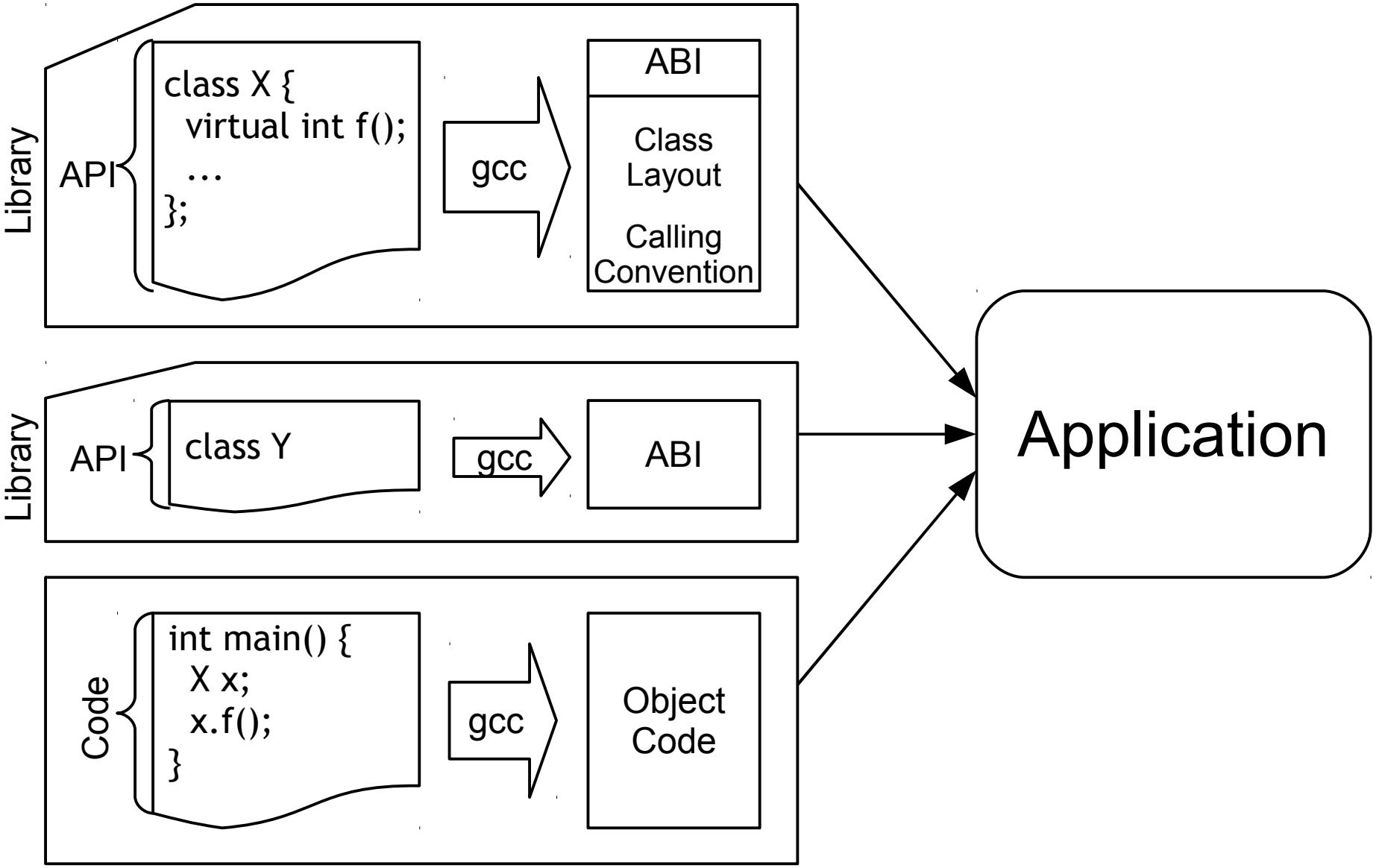
Kevin Atkinson

PhD Thesis
*University of Utah*

# Application Programmer Interface (API)

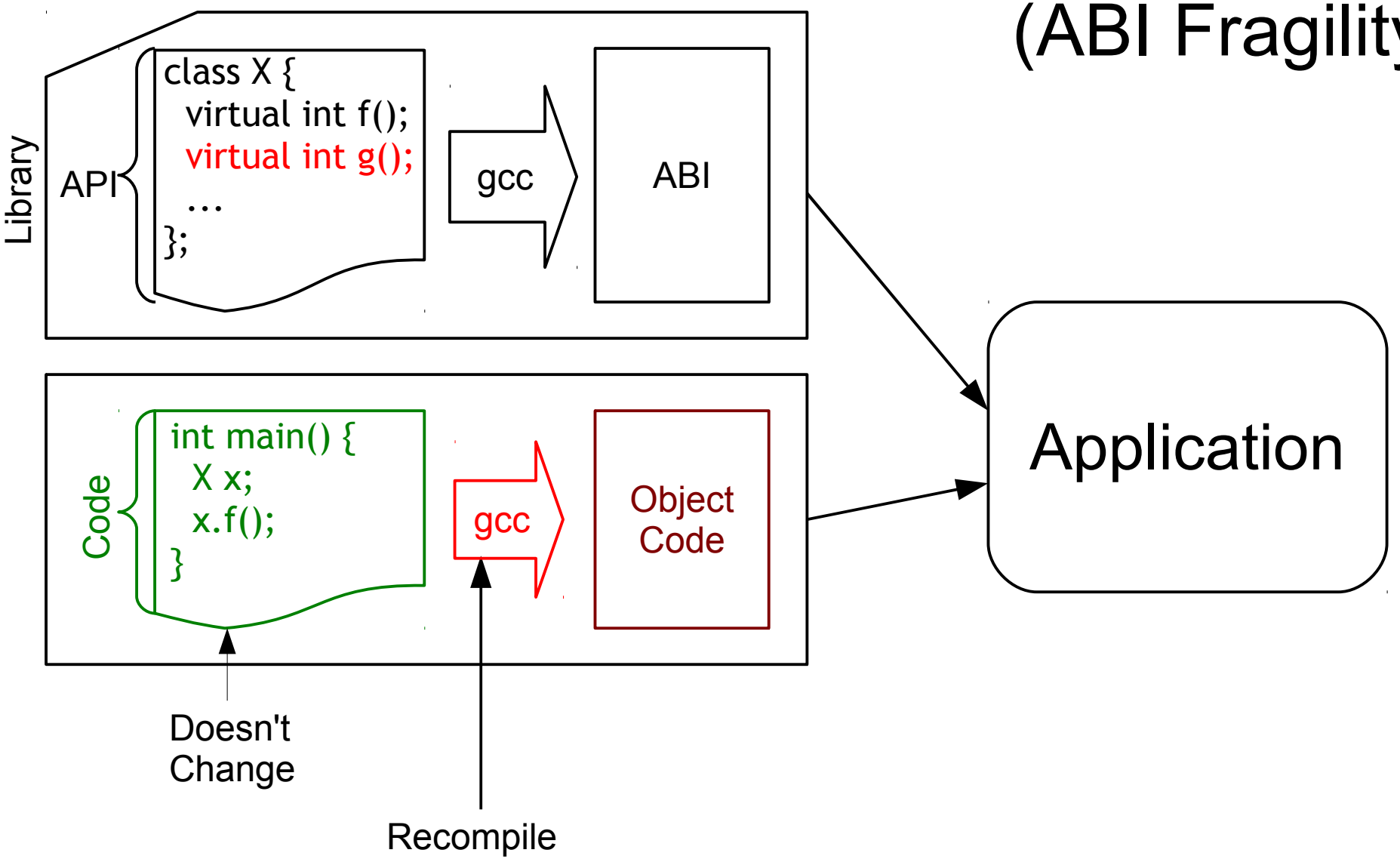


# Application Binary Interface (ABI)

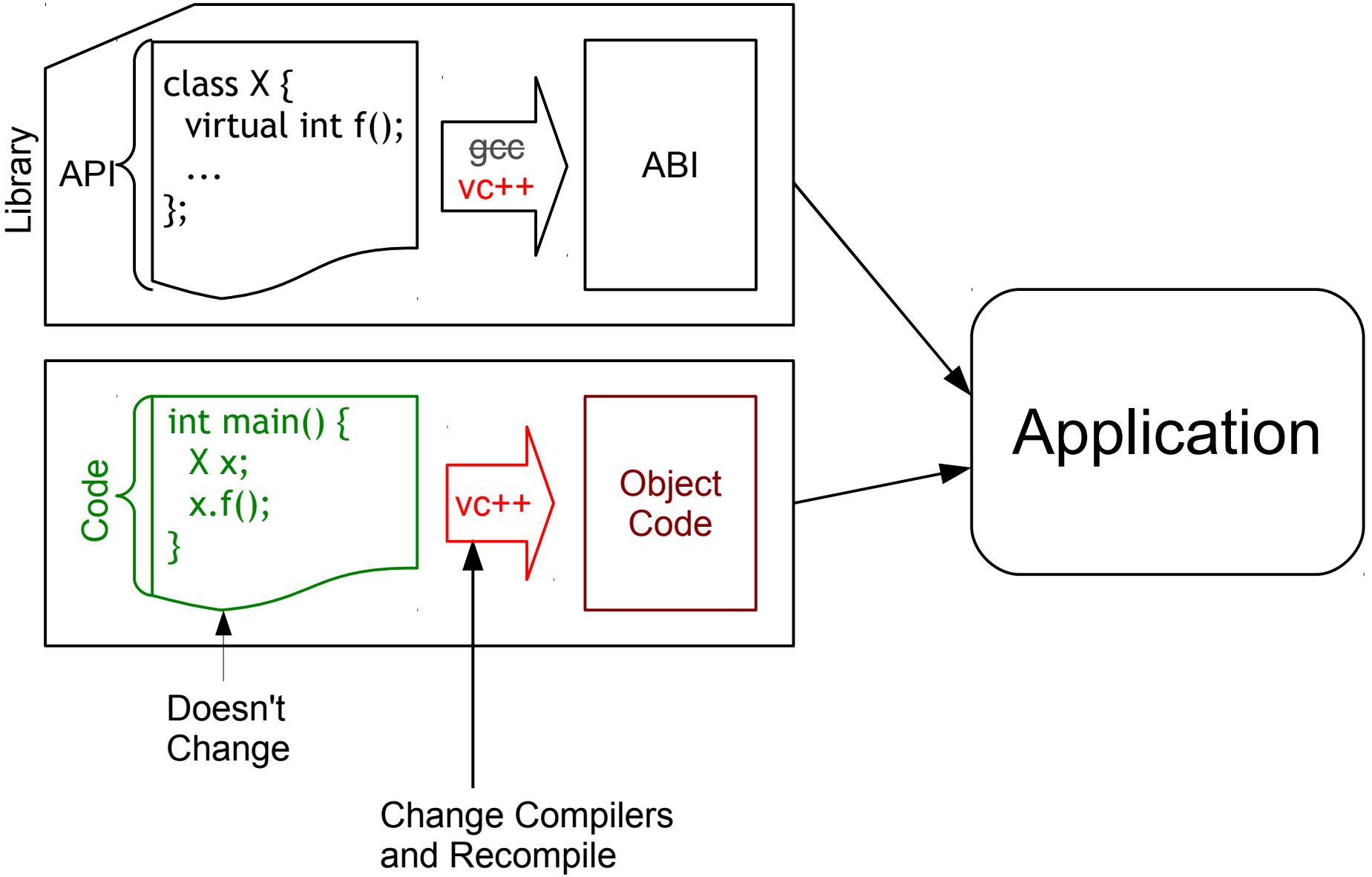


# Source vs Binary Compatibility

## (ABI Fragility)



# ABI Incompatibility



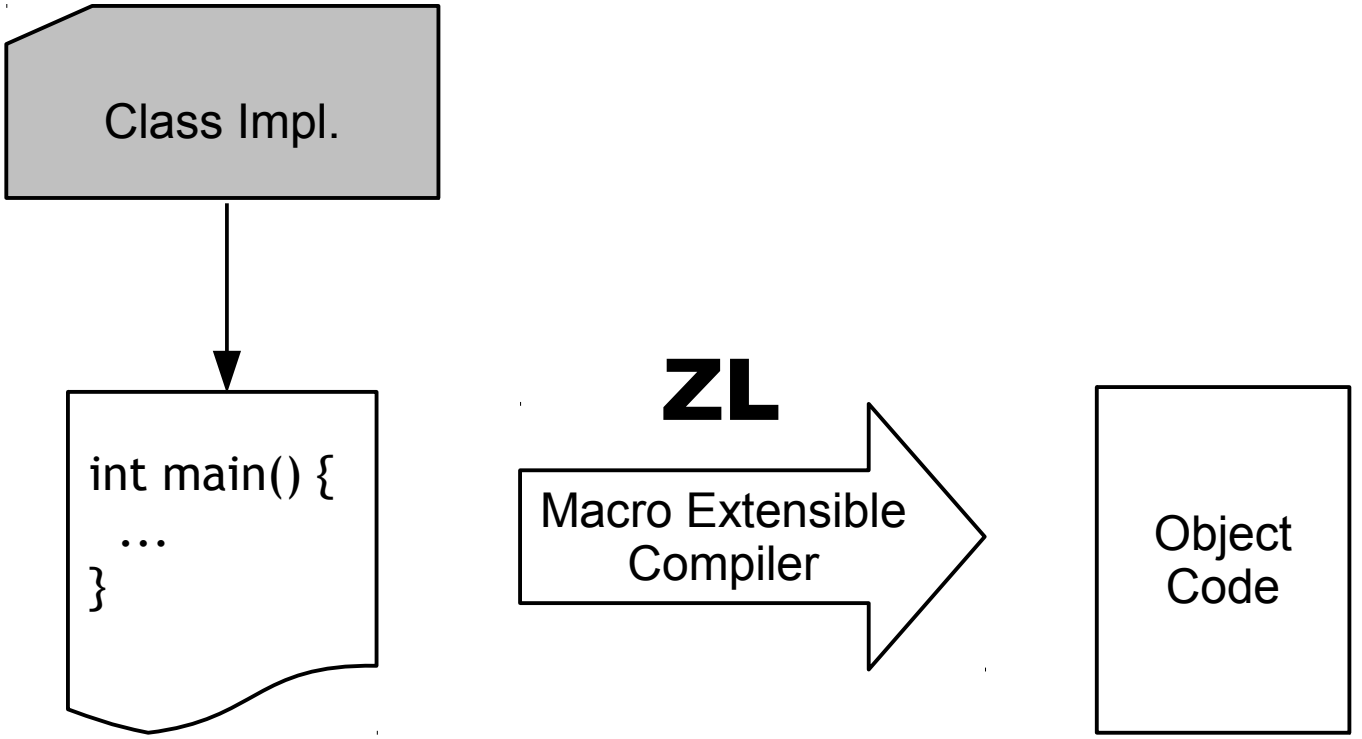
# Previous Work

- Some Standardization Attempts
  - Itanium C++ ABI
- Still Two Common ABIs:
  - GCC
  - Visual C++

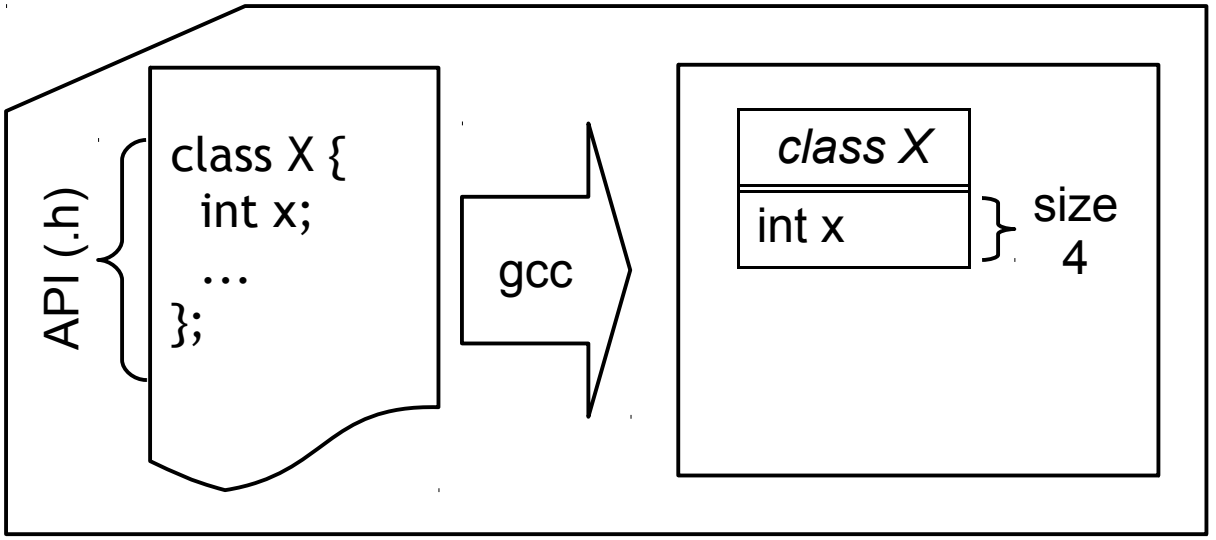
- Some Work Towards Less Fragile ABIs
  - Delta C++ (Palay, 1992)
  - Object Binary Interface (Williams and Kindel, 1994)
- Sacrifice Performance
- Not Commonly Used

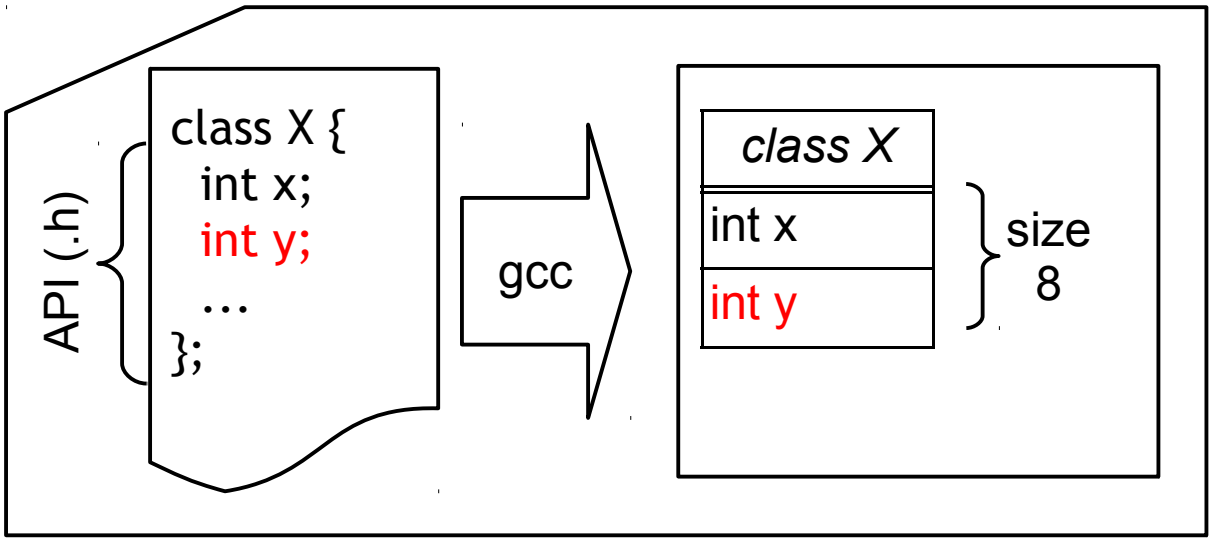
# Thesis Statement

Fragile and incompatible ABIs are a major problem in software maintenance and evolution that can be systematically dealt with and effectively managed using a macro-based system that allows the programmer to control how an API maps to an ABI.



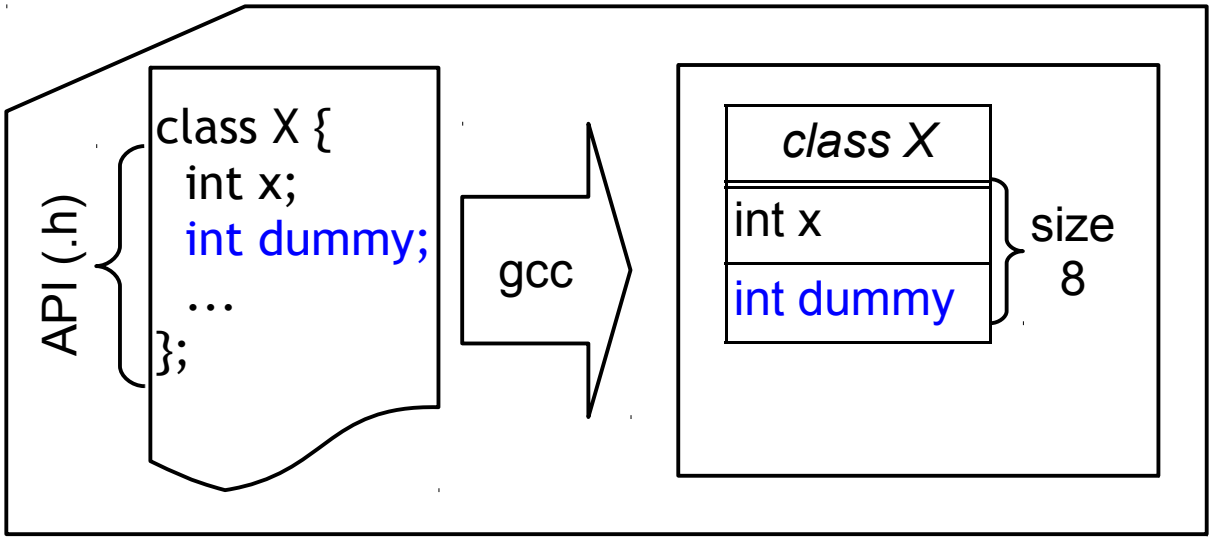
# Breaking Binary Compatibility

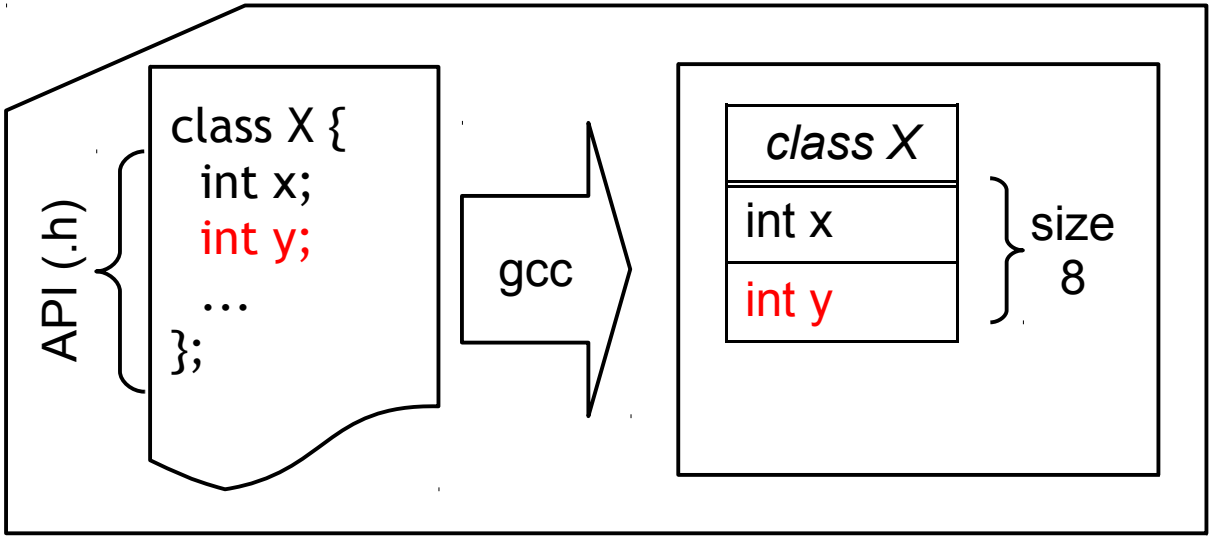


# Preserving Binary Compatibility

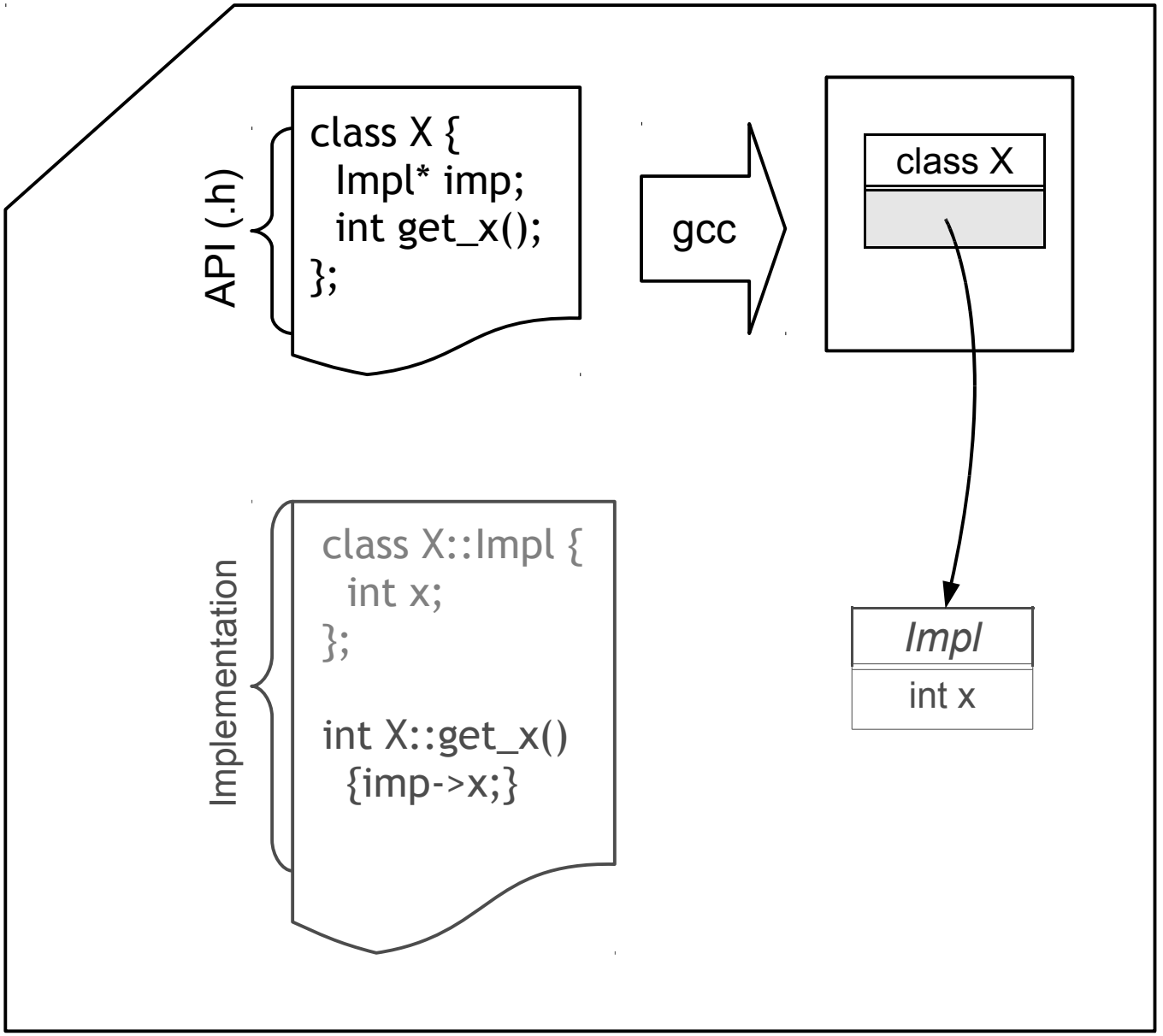
Dummy Member



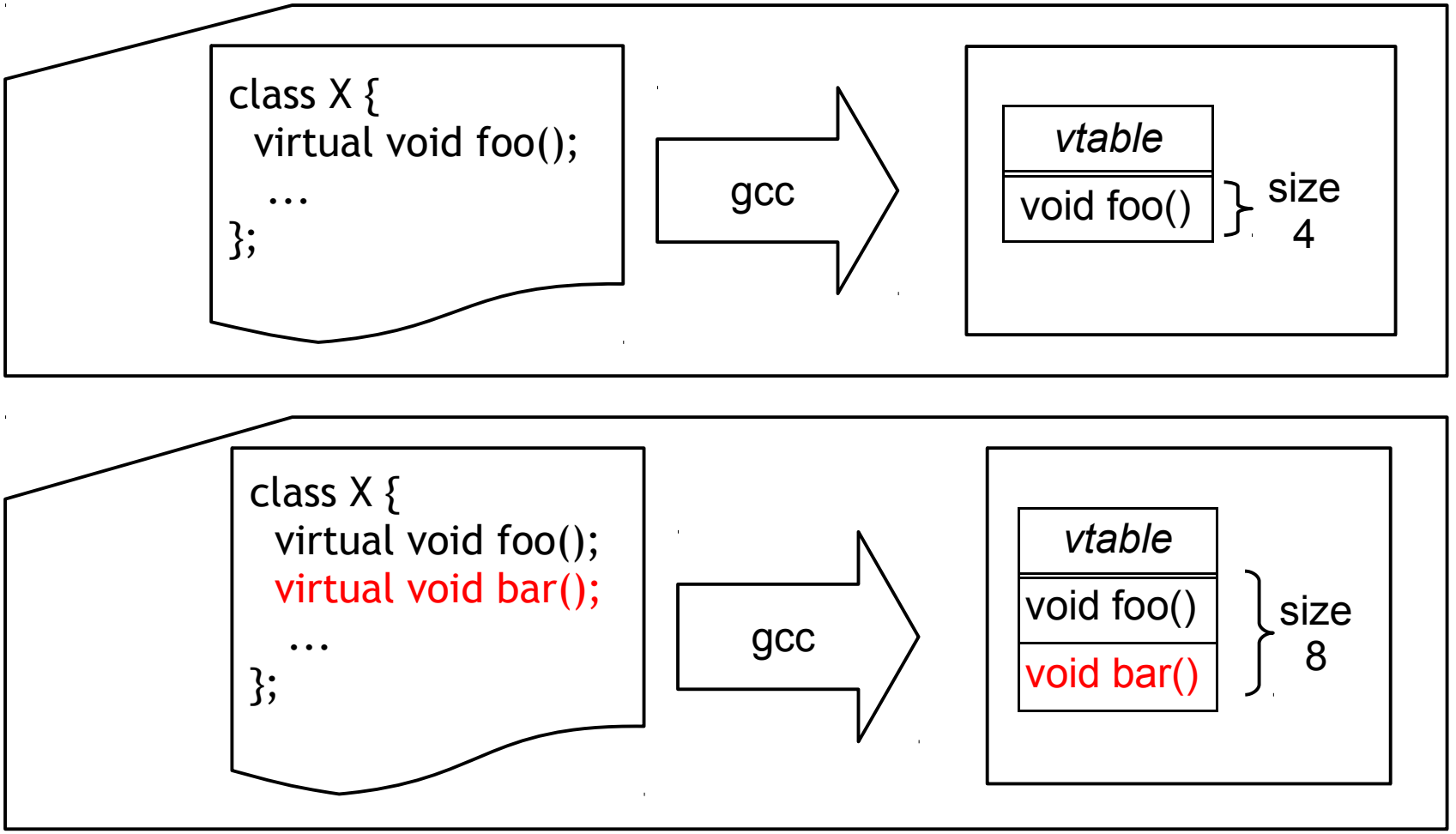



# Preserving Binary Compatibility

Pointer to Implementation

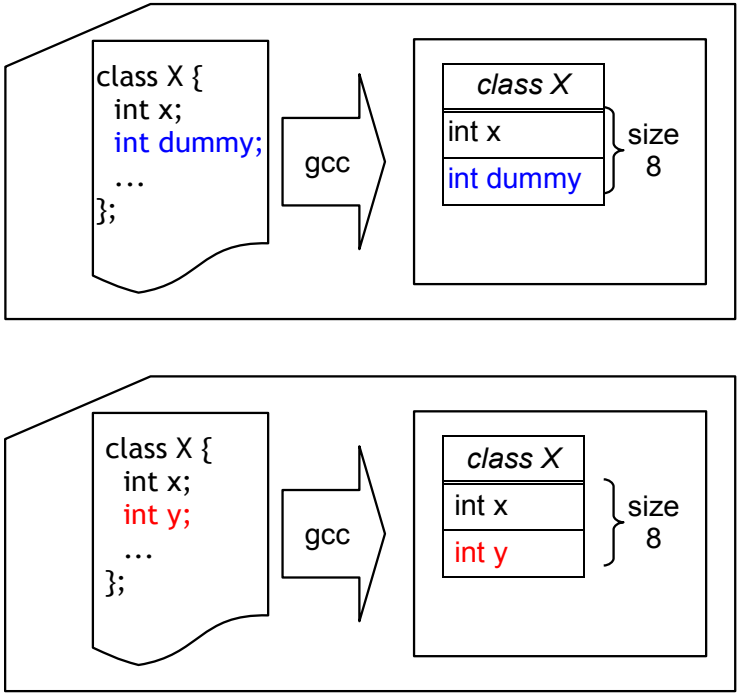


# Breaking Binary Compatibility

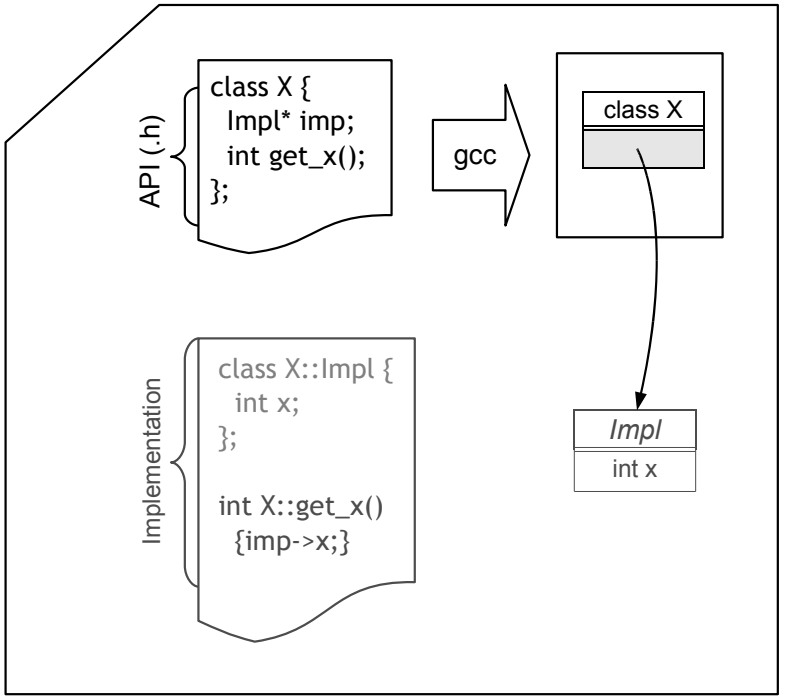



- For Preserving: Options Limited

## Dummy Member

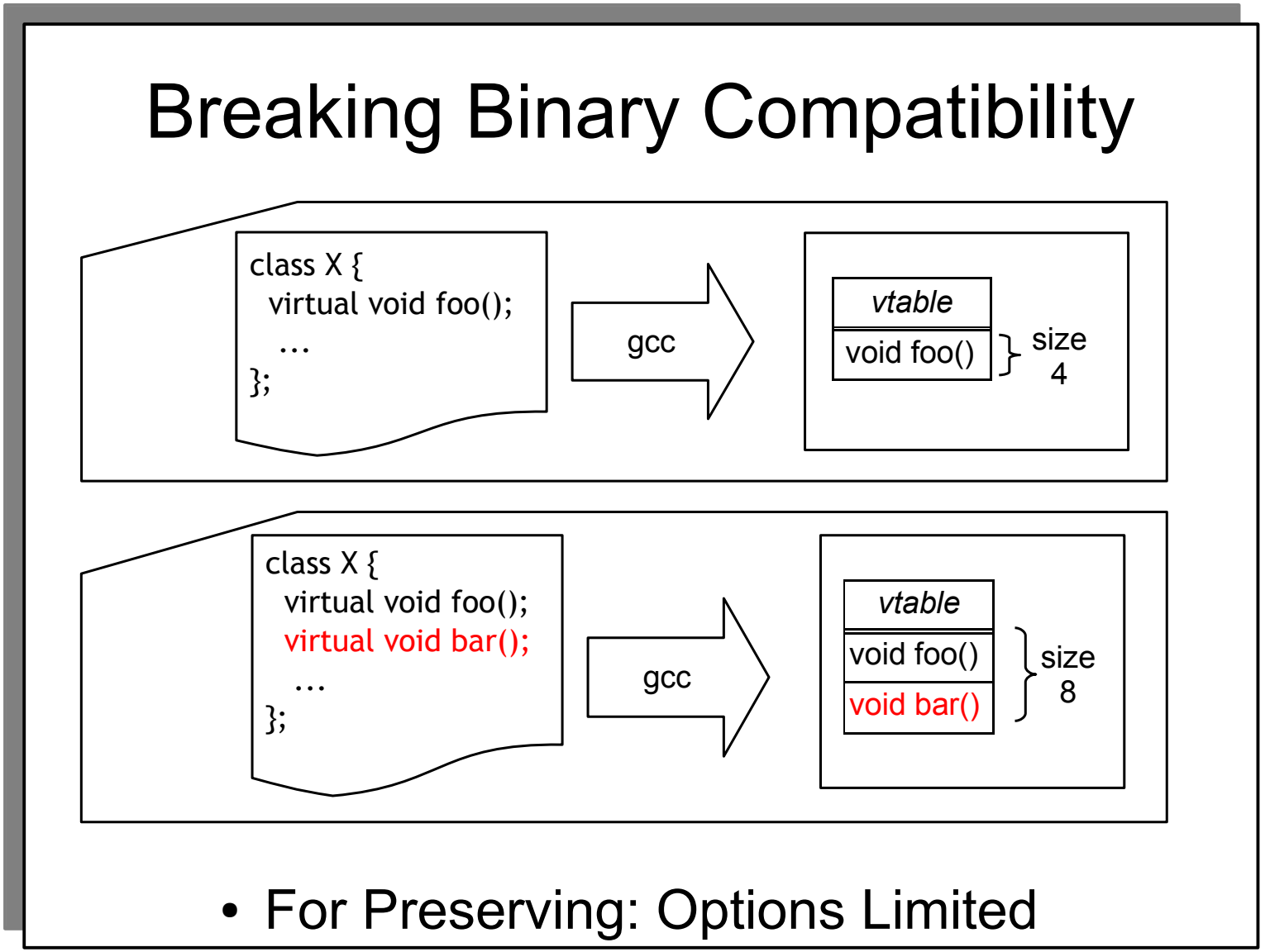


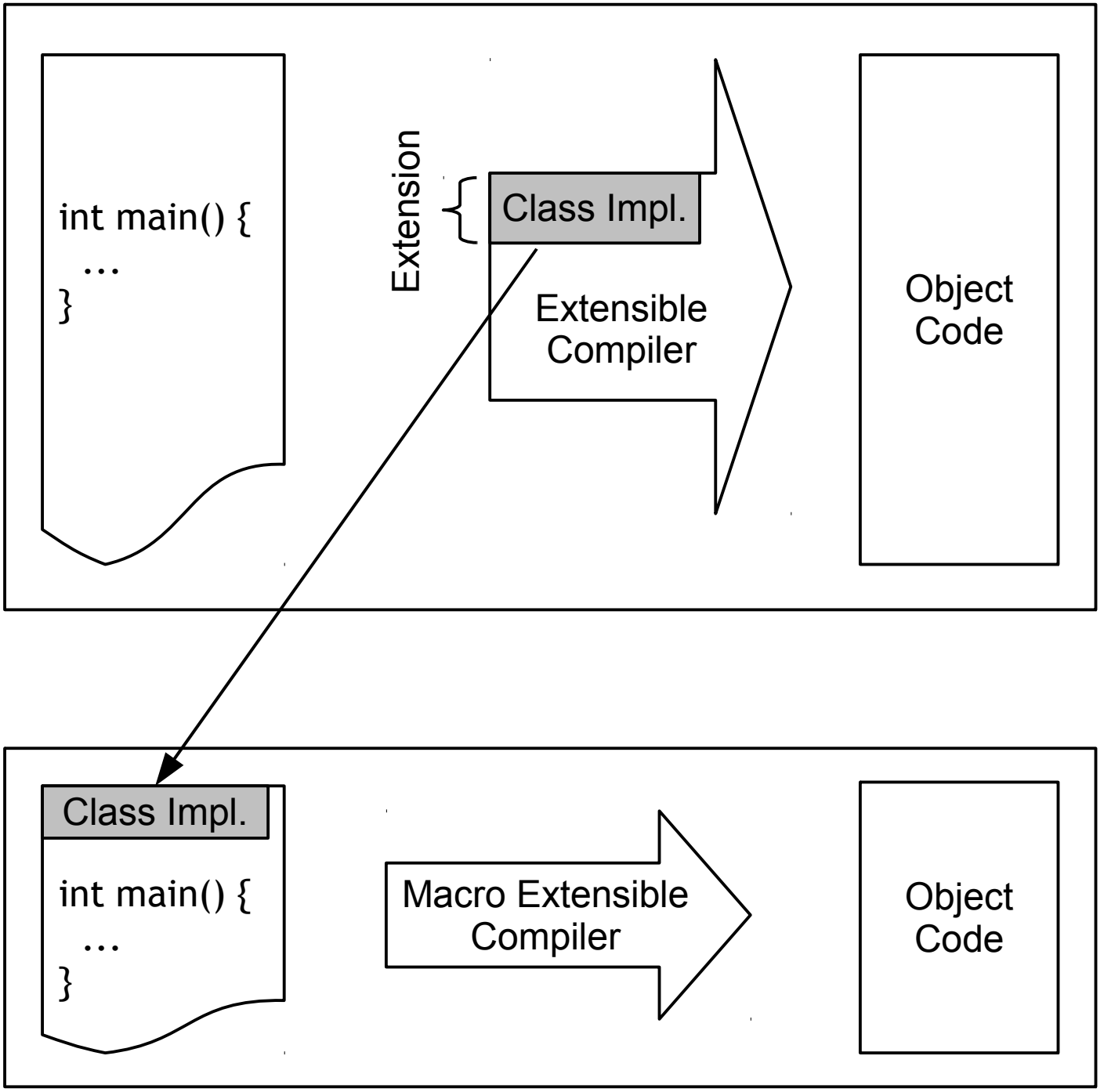

## Pointer to Impl.

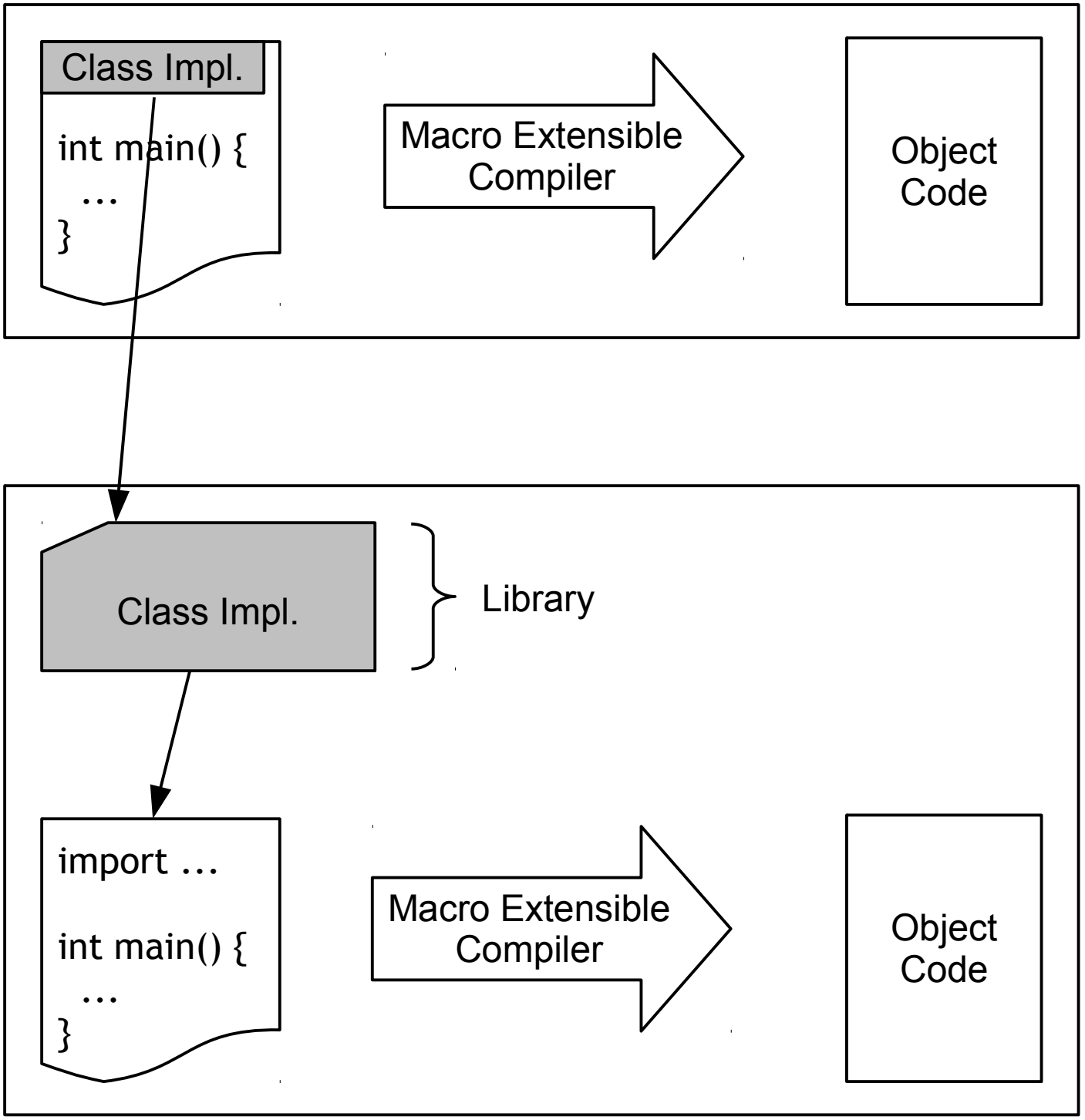



Automate These Patterns

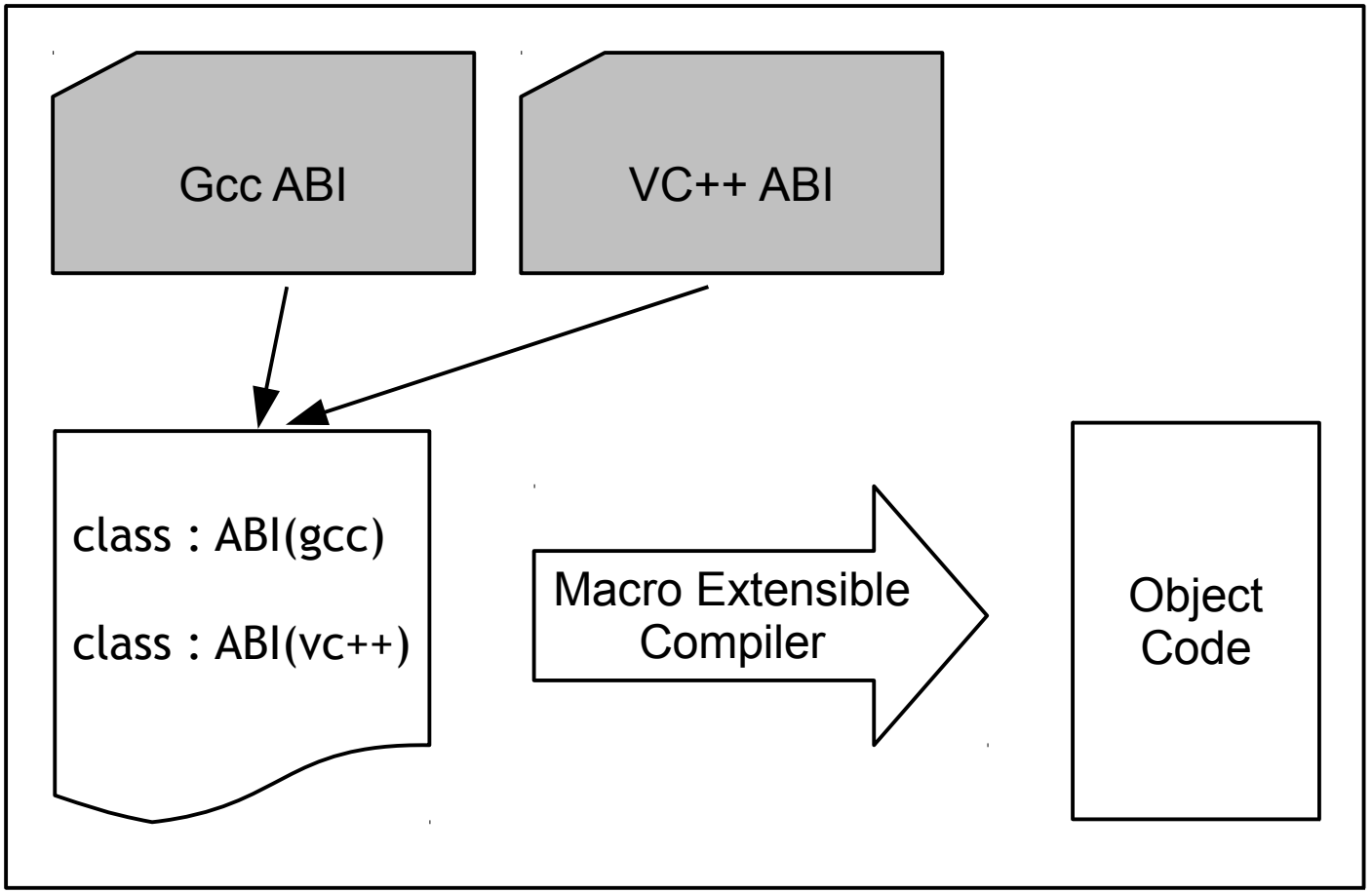
# Automation is Not Enough...



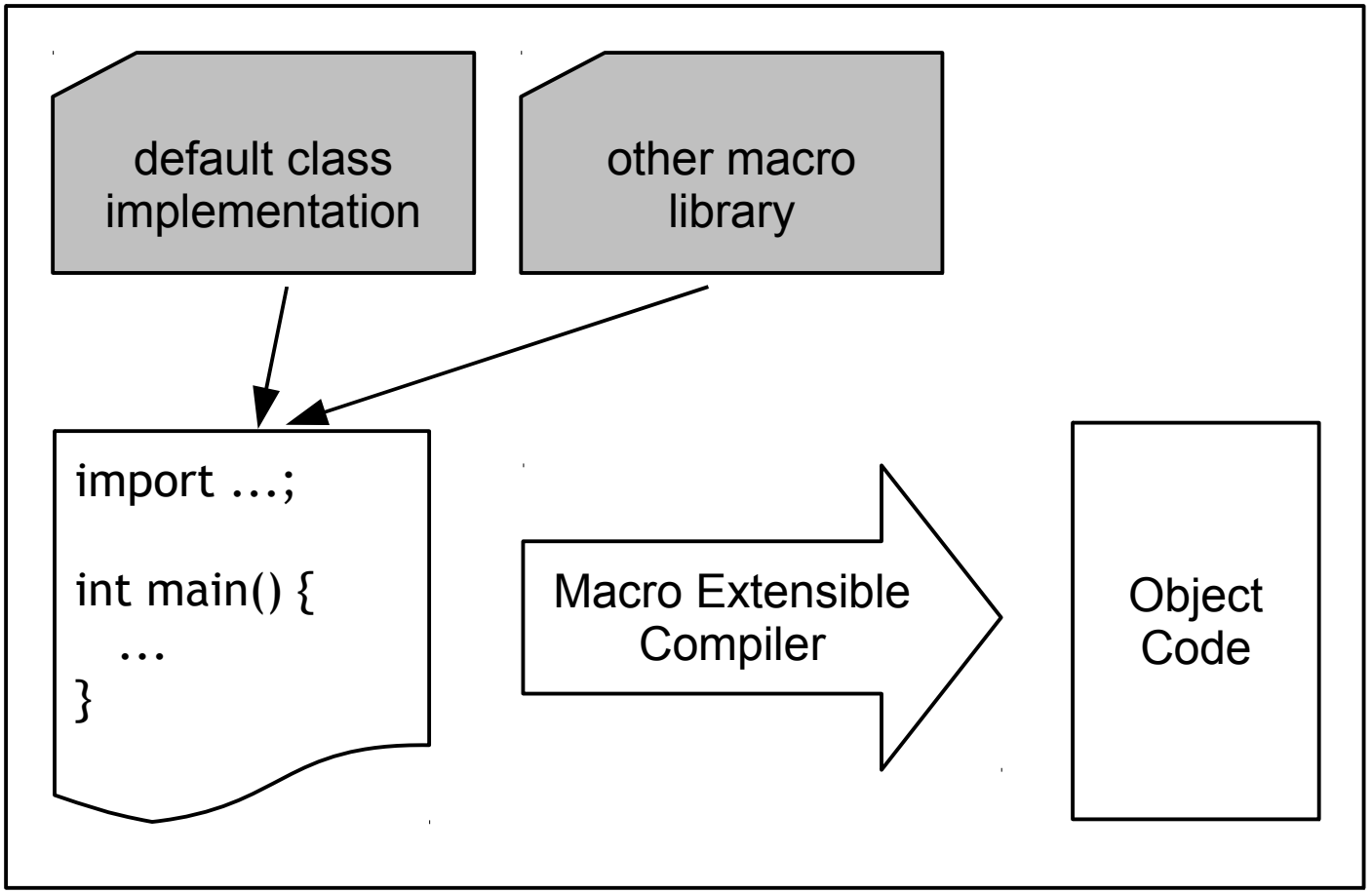

int main() {
…
}
Extension
Class Impl.
Extensible
Compiler
Object
Code
Class Impl.
int main() {
…
}
Macro Extensible
Compiler
Object
Code


Class Impl.
int main() {
…
}
Macro Extensible Compiler
Object Code
Class Impl.
Library
import …
int main() {
…
}
Macro Extensible Compiler
Object Code

# Two ABI's at Once



# ZL



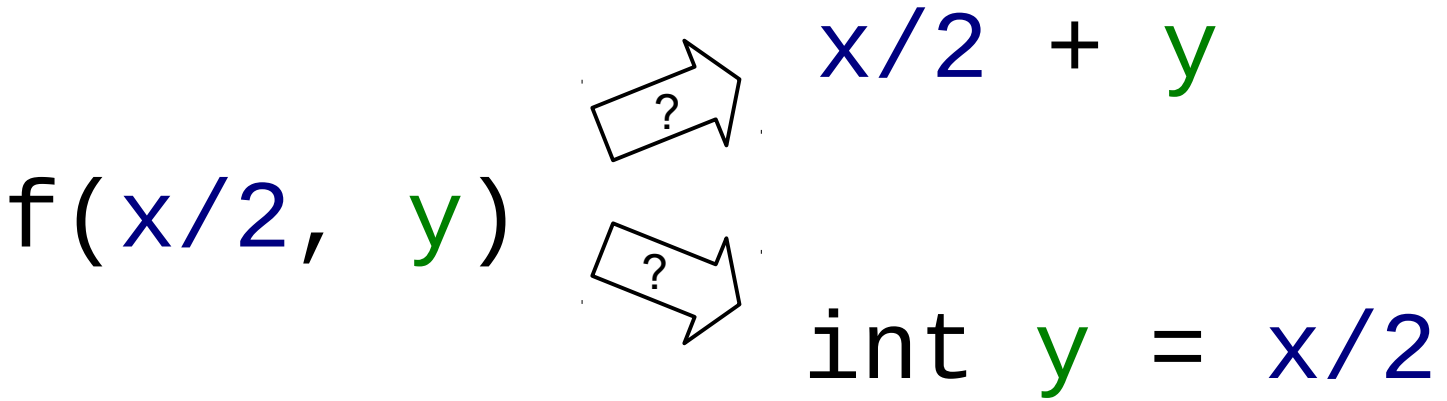

C Like Core w/ Minimal Amount of C++

# How to Parse This Expression?

```
f(x/2, y)
```

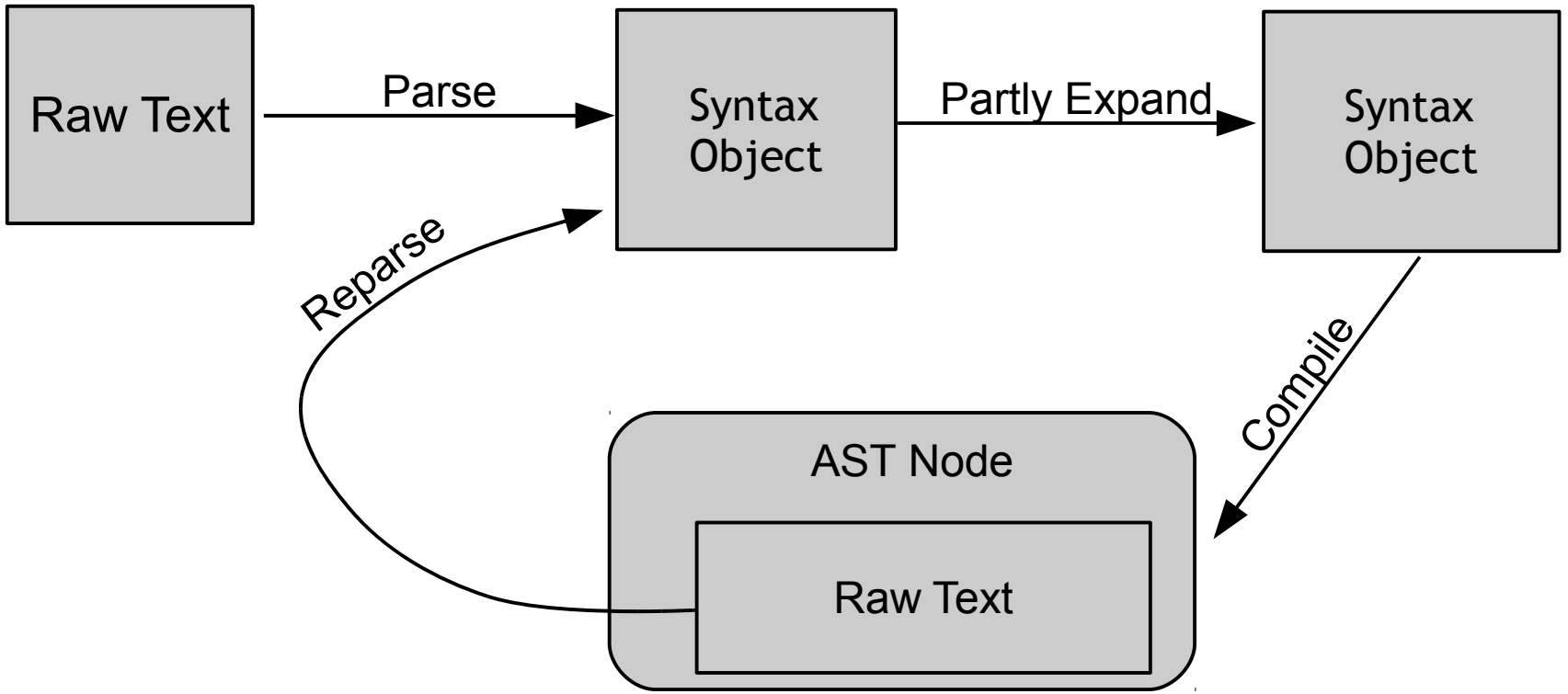
Function Call?

Macro Invocation?



# Parsing Overview

- ZL doesn't parse in a single linear pass
- Iterative-deepening approach



# Parsing Details

```
inline int f() {int x = 10; return x;}
int main() {return f();}
```

Parse

```
(@ (stmt inline int f ("()" "")
                      ("{}" "int x = 10; return x;")
   (stmt int main ("()" "") ("{}" "return f();")))
```

Expand & Compile

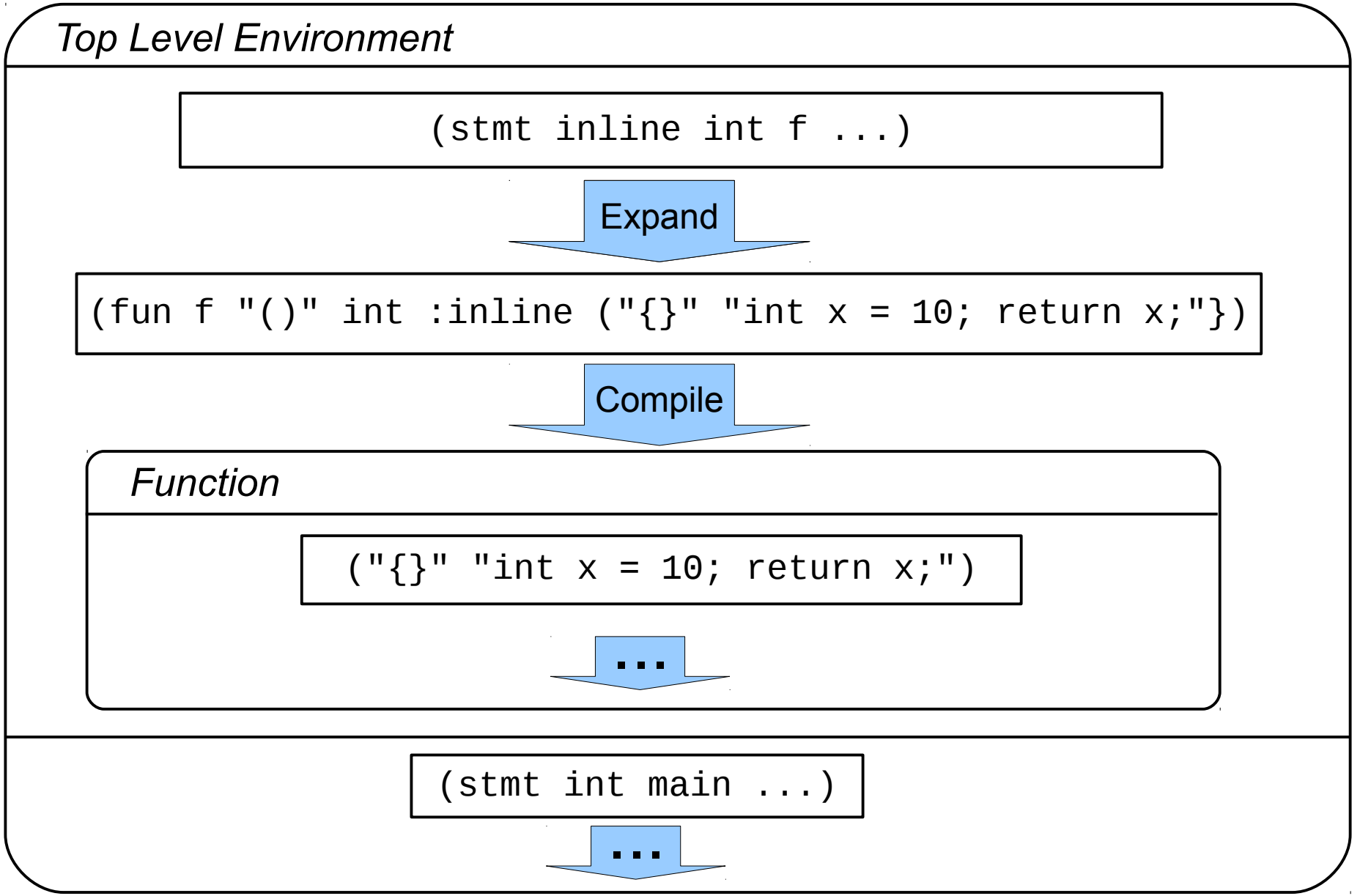
*Top Level Environment*

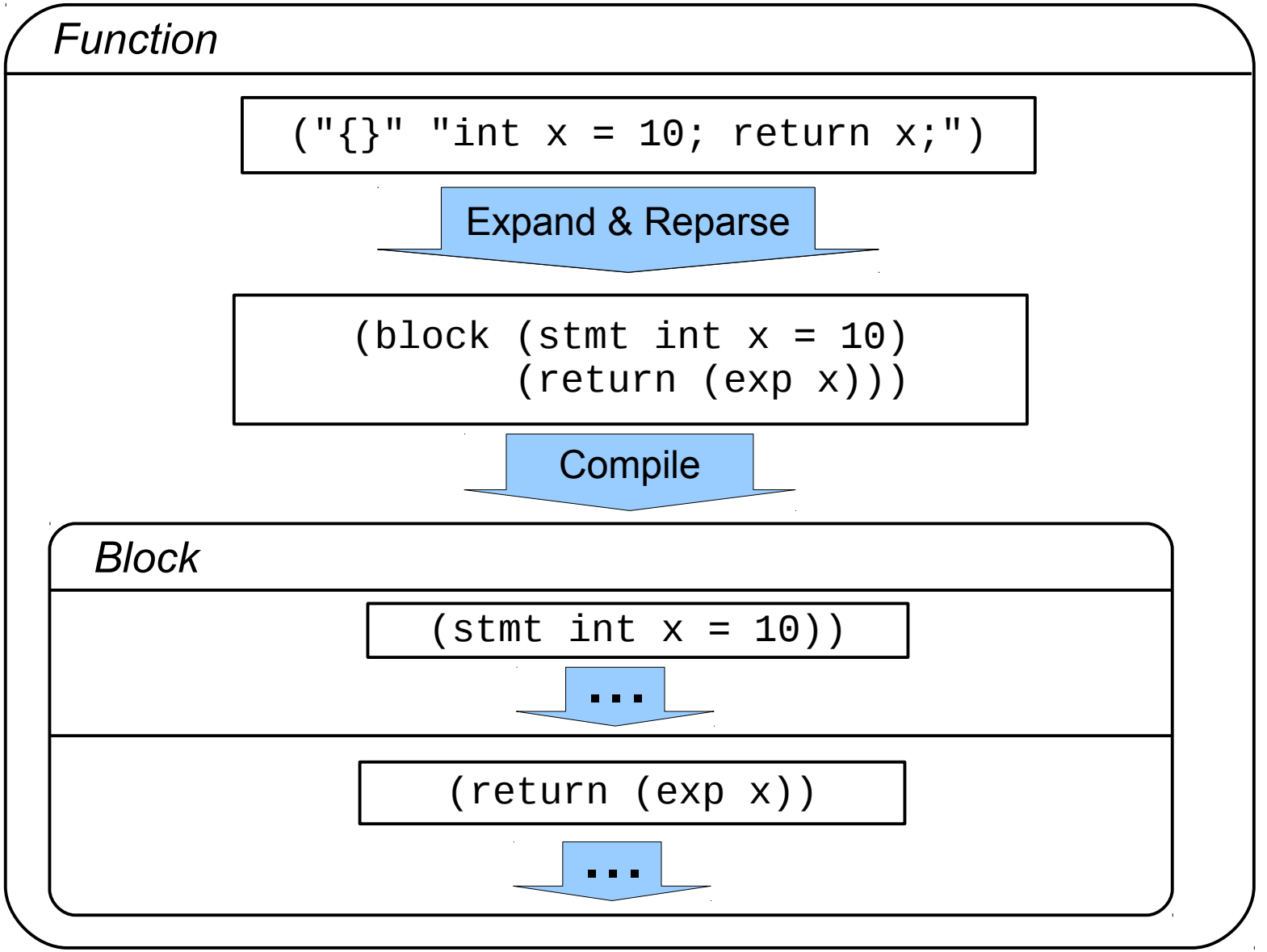
```
(stmt inline int f ...)
```

...

```
(stmt int main ...)
```

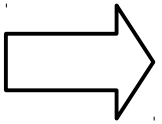
...


Top Level Environment
(stmt inline int f ...)
Expand
(fun f "()" int :inline ("{}" "int x = 10; return x;"})
Compile
Function
("{}" "int x = 10; return x;")
...
(stmt int main ...)
...


Function
("{}" "int x = 10; return x;")
Expand & Reparse
(block (stmt int x = 10)
(return (exp x)))
Compile
Block
(stmt int x = 10))
...
(return (exp x))
...

# Hygienic Pattern Macros

**macro** or(*x*, *y*) { ({typeof(*x*) t = *x*; t ? t : *y*;}); }

or(0.0,t)  ({typeof(0.0) $t_0$ = 0.0; $t_0$ ? $t_0$ : t;});

# Procedural Macros

```
Syntax* foreach(Syntax*) {...}
make_macro foreach

int main() {
  Container c;
  ...
  foreach(el, c, el.print());
}
```

*Syntax Forms:*
- new_mark
- syntax
- make_macro

*Callbacks:*
- match
- replace
- ct_value
- error

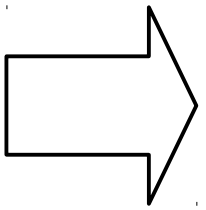
# The Class Macro

```
import_file "class-simple.zlh";

...

.class ParseClass {
  ...
  virtual Syntax * parse(Syntax * p) {...}
  virtual void init() {...}
  virtual void init_parent_info() {...}
  ...
};

Syntax * parse_class(Syntax * p) {
  ParseClass * pc = new ParseClass;
  return pc->parse(p);
}
make_macro class parse_class;
```

900 Lines of Code

# Expanding Method Calls

```
C o
o.f()
```



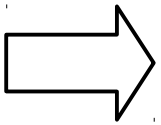
```
C::f(...)
```

What is f?

# User Types
## (ZL's Notion of Classes)

```
user_type C {

  struct Data {
    int i;
  };
  associate_type struct Data;

  int foo(int j)

  macro i(:this ths) {...}
  macro f(x, :this ths) {...}
}
```

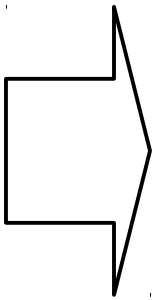
```
void main() {
  C o;
  o.f(x);
  int x = o.i;
}
```



```
void main() {
  C o;
  C::f(x, :this = &o);
  int x = C::i(:this = &o);
}
```

# Classes

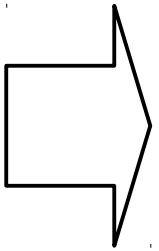
```
class C {

  int i;

  int f(int j)
    {return i + j;}

};
```



```
user_type C {

  struct Data {
    int i;
  };
  associate_type struct Data;
  int f`internal(C * this, int j)
    {return i + j;}


  macro i (:this ths = this)
    {(*(C *)ths)..i;}
  macro f(j, :this ths = this)
    {f`internal(ths, j);}
}
```

# Inheritance

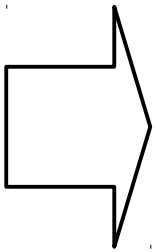
```
class D : public C
{

 int j;

};
```



```
user_type D {

  import C;

  struct Data {
    struct C::Data parent;
    int j;
  };
  associate_type struct Data;

  make_subtype C _up_cast _down_cast;
  macro _up_cast ...
  macro _down_cast ...

  macro j (:this ths = this) {...}

}
```

# Virtual Methods

```
class D : public C
{
  virtual
    void f(int j);
};
```

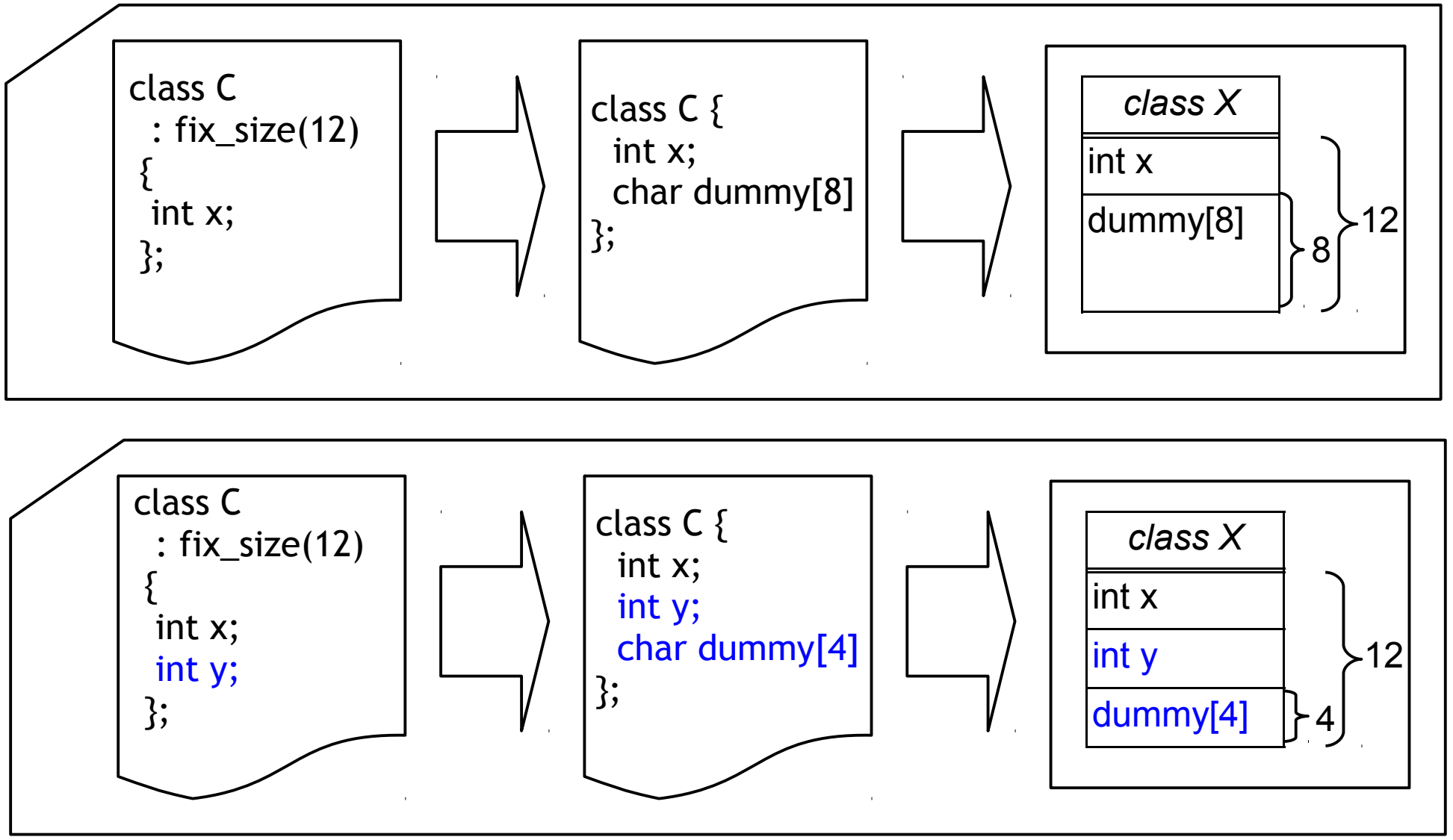


```
user_type D {
  ...

  class VTable : public C::VTable {
    void (*f)(int j);
  };

  struct Data {
    VTable* _vptr;
    ...
  };
  associate_type struct Data;

  macro f(j, :this ths = this)
    {_vptr->f(ths, j);}

  ...
}
```

# Examples

## Using ZL to Mitigate ABI Problems

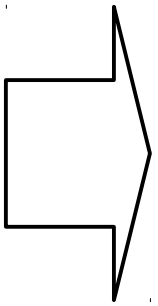
# Example: Fixing Size of Class

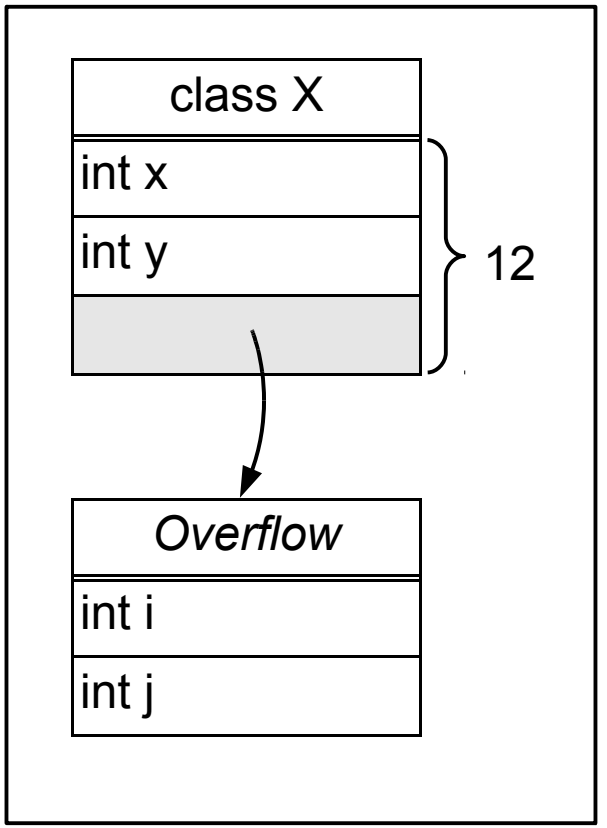



(Complete Macro In Dissertation, 36 Lines of Code)

# Example: Allowing More Fields

```
class C :
  fix_size(12)
{
  int x;
  int y;
  int i;
  int j;
};

int main() {
  C obj;
  obj.x;
  obj.i;
}
```

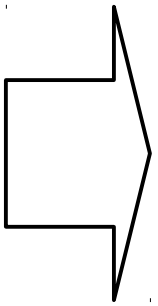


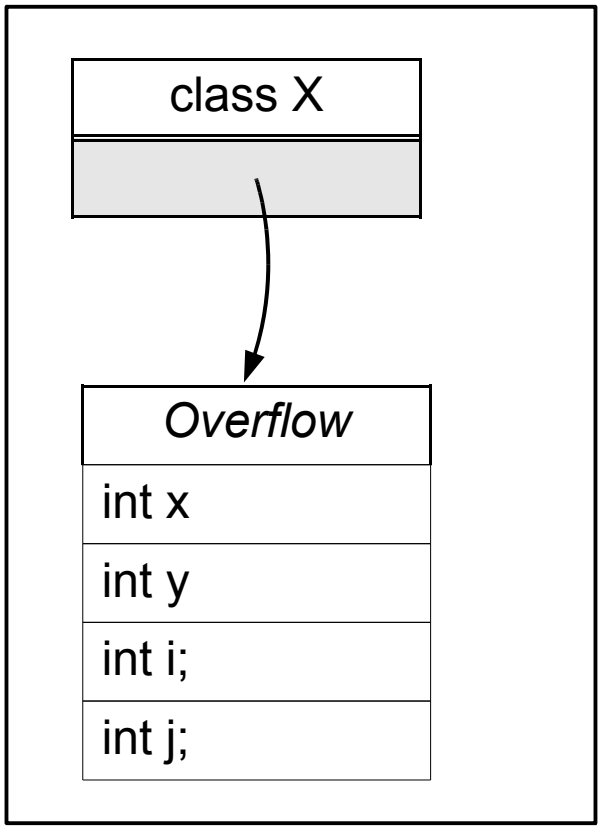



(Expanded Macro, Around 100 Lines of Code)

# Example: Pointer to Impl.

```
class C :
  fix_size(0)
{
  int x;
  int y;
  int i;
  int j;
};

int main() {
  C obj;
  obj.x;
  obj.i;
}
```

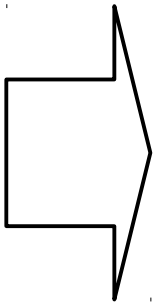





(Same Macro, Around 100 Lines of Code)

# Fixing Virtual Table Size

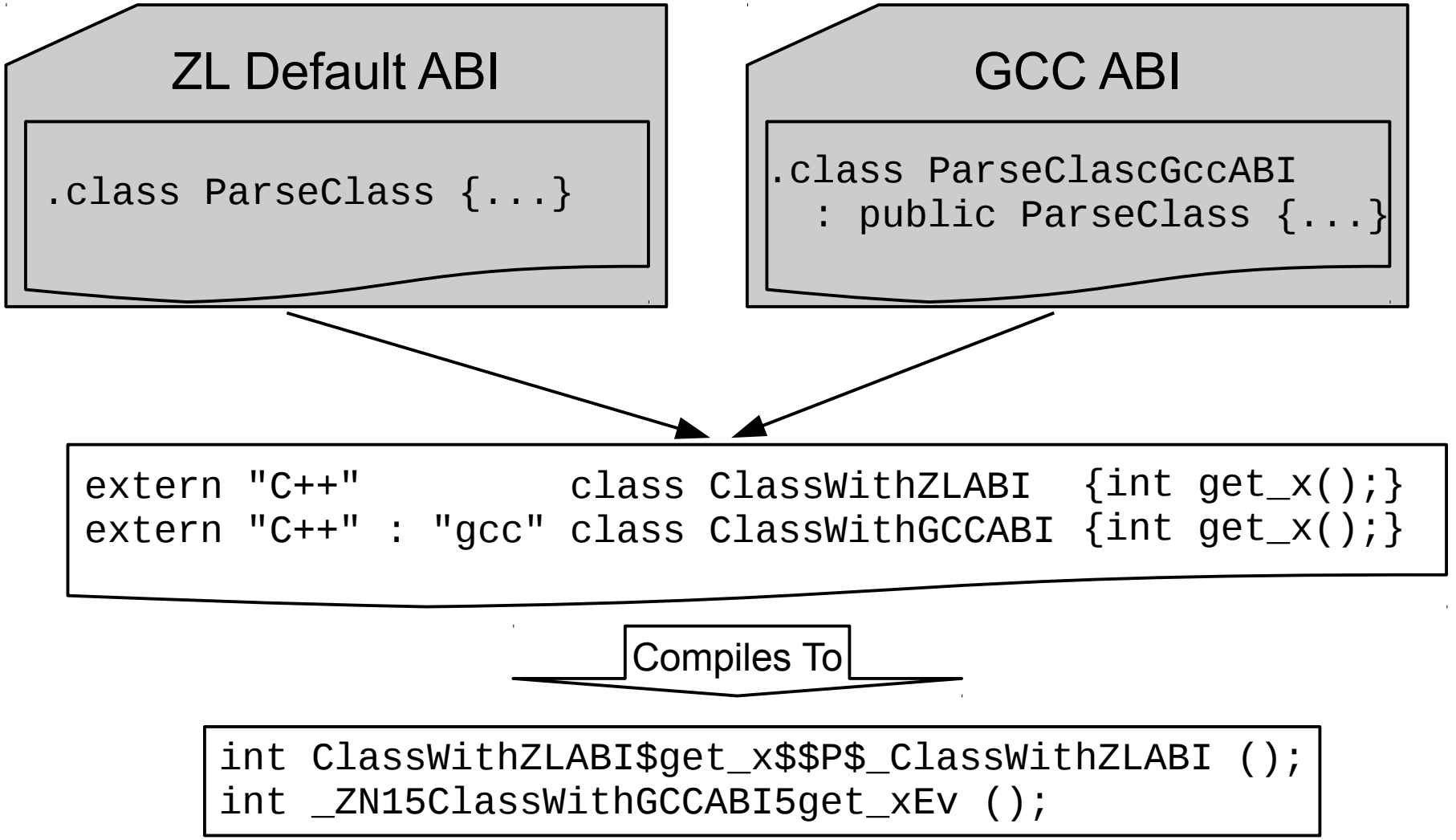
```
class X
  : fix_vtable_size(8)
{
  virtual void f();
  virtual void g();
  virtual void h();
};
```



```
macro alt_vtable (name, body) {
  class name : fix_size(8) {body};
}

class X : vtable_class(alt_vtable) {
{
  virtual void f();
  virtual void g();
  virtual void h();
};
```

(Modified Macro, Around 120 Lines of Code)

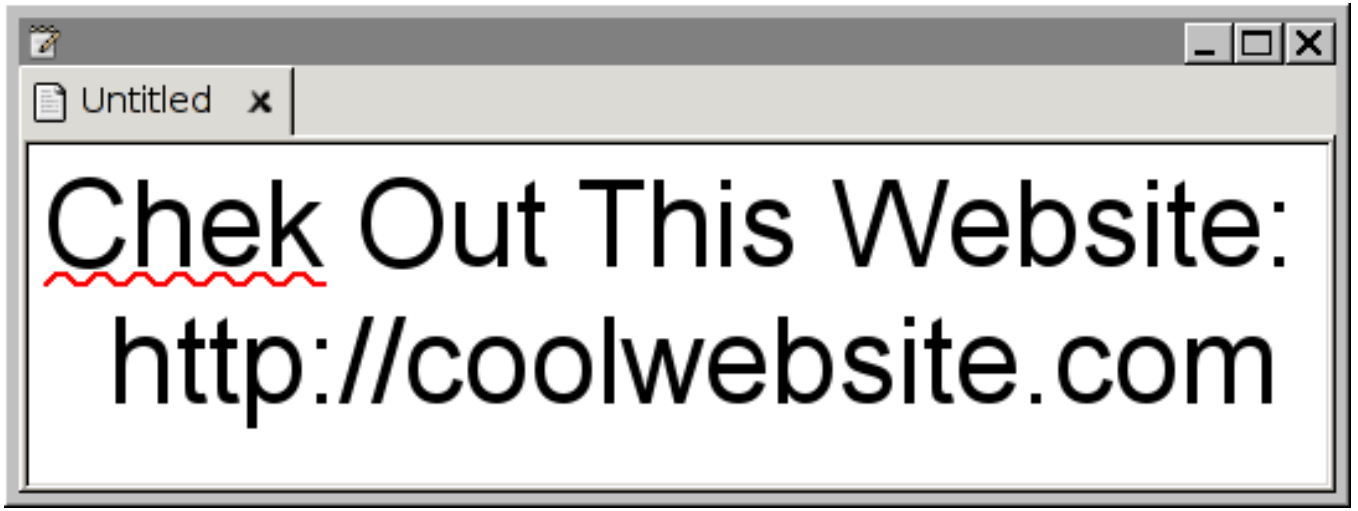
# Matching GCC's ABI



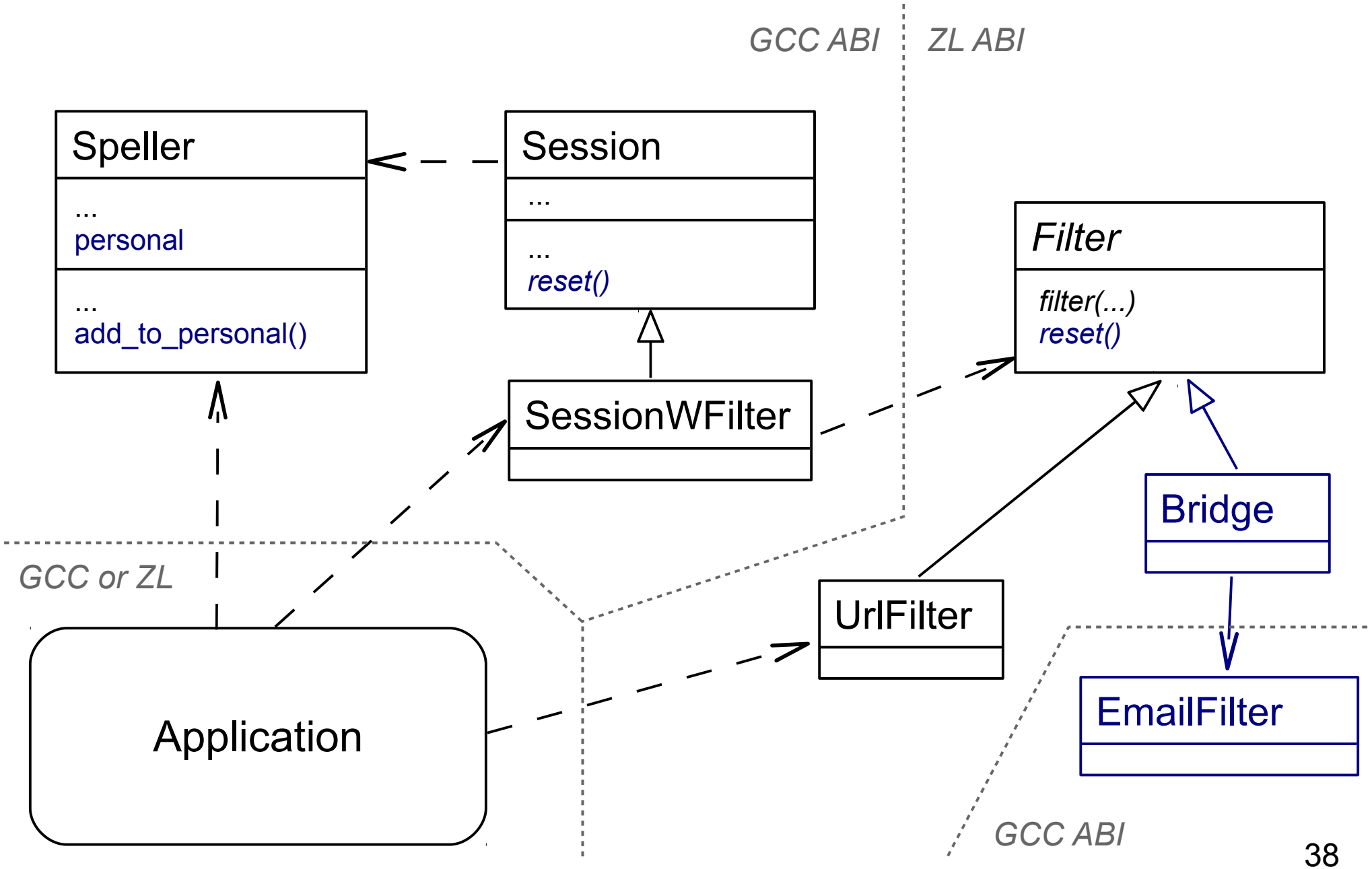

(Class Macro 150 lines, Mangler 300 lines)

# Demonstration

*Simple Spell*



# Simple Spell´s Classes



# Speller Class

```
...
Speller * speller = new Speller();
speller->init(new_lang(), new_master_dict());
...
for (...) { // look through document
    bool correct = speller->check(word);
    if (!correct) {
        // maybe
        Suggestions * sugs = speller->suggest();
        // if indeed correct
        speller->add_to_session(word);
    }
}
...
```

*GCC ABI*

```
class Speller {
    init(...)
    check(...)
    suggest()
    add_to_session(...)
};

struct Suggestions;

new_lang();
new_master_dict();
```

# Session Class

```
...
Speller * speller = new Speller();
speller->init(new_lang(), new_master_dict());
SessionWFilters * session
    = new SessionWFilters(speller);
session->add(new_url_filter());
for (...) { // look through doc. one line at a time
   session->new_line(line);
   while (session->next_misspelling()) {
       word = session->misspelled_word();
       bool correct = speller->check(word);
       if (!correct) {
         // maybe
           Suggestions * sugs = speller->suggest();
           // if indeed correct
           speller->add_to_session(word);
        // if incorrect
        session->replace(new_word)
       }
}
...
```

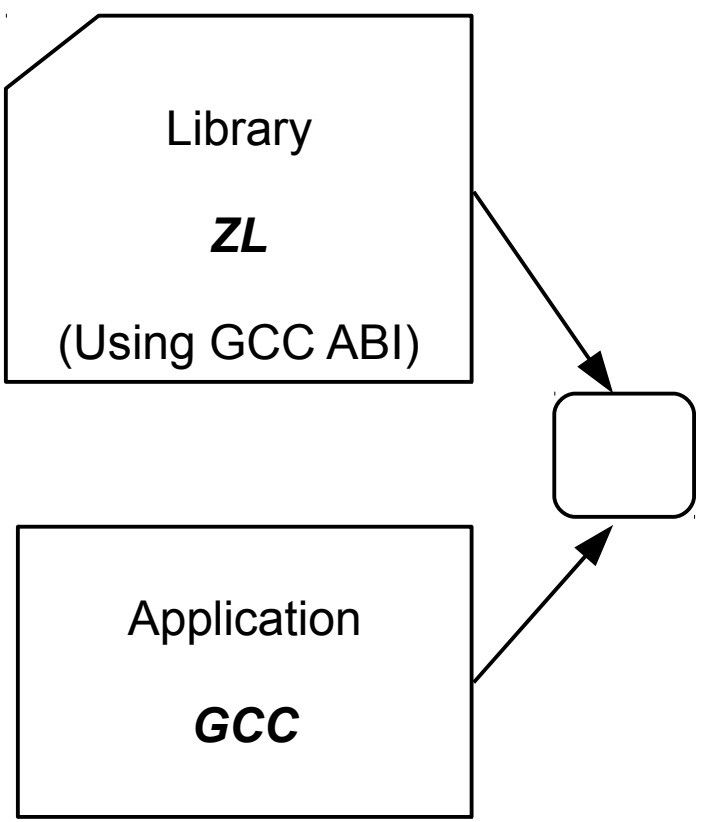
*GCC ABI*

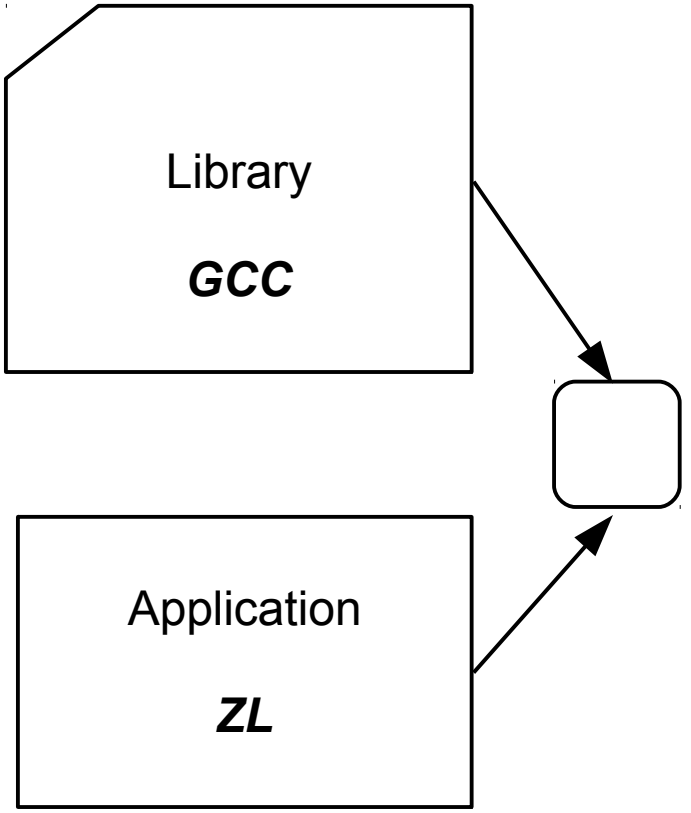
```
class Session {
  virtual new_line(...)
  virtual next_misspelling()
  inline mispelled_word()
  virtual replace(...)
};
class SessionWFilters
  : public Session {
      virtual add(Filter *)
};
Filter * new_url_filter();
```

# Extension API

# Simple Application

```
*Teh* dog swm up the stream.
1) The  2) Tea  3) Ted  4) Tee  5) Tel  6) Ten ...
i) Ignore  I) Ignore all  r) Replace  a) Abort
```



# Email Filter with GCC ABI

*GCC ABI*

```
class EmailFilter
          : public Filter
  {...}
Filter * new_email_filter();
```

```
> This line skippd
This line checkd
```

*ZL ABI*

```
class FilterBridge : public Filter {...}

Filter * filter_bridge(/* filter in GCC ABI*/)
```

```
session->add(filter_bridge(new_email_filter()));
```

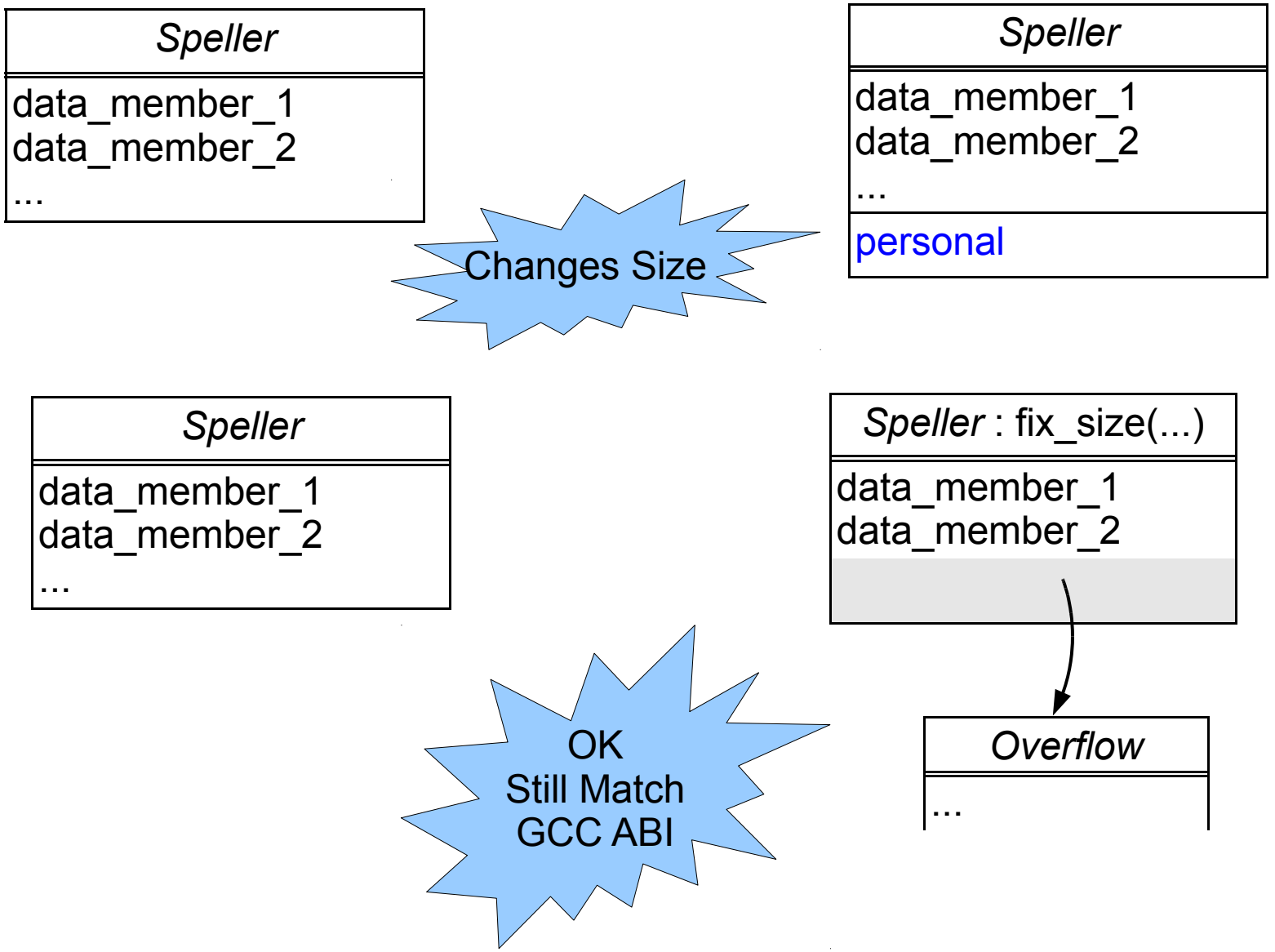
# Automate Bridge Class

```
Syntax * parse_mk_bridge(...) {
    ...
}
make_macro mk_bridge parse_mk_bridge();
```

```
mk_bridge(Filter, filter_bridge, “GCC”)
```

(Macro Just Under 55 Lines of Code)

# Support For A Personal Dictionary



# Fixing Size of Speller Class

```
class SpellerOld {
  // original Speller class private data members
};

class Speller
  : fix_size(sizeof(SpellerOld))
{
  // private data members
  SavableDict * personal;
  // original methods
  void add_to_personal(const char *);
  void save_personal();
};
```

# An ABI Allowing Future Adaptation

```
class Speller
   : fix_size(0) {
  ...
};
```

```
extern “C++” : “better”
class Filter {
   ...
}
```

```
class Session
  : fix_size(/* size of members used
                by inline functions */)
  : vtable_slots(16)
{
  const char * word
  ...
  const char * misspelled_word()
    {return word;}
  ...
};

class SessionWFilter
  : fix_size(0)
  : vtable_slots(16) ...
```

# Simple Spell, Version 2

## Add (Back) Support for a Personal Dictionary

```
class Speller : fix_size(0) {
  ...
  SavableDict * personal;
  ...
  void add_to_personal(...);
  void save_personal();
};
```

# Simple Spell, Version 2

## Add Support for Stateful Filters

```
class Session
   : fix_size(...)
   : vtable_slots (16)
{
   ...
   void reset();
};
```

```
extern “C++” : “better”
class Filter {
    ...
    void reset()
}
```

```
Filter * new_comment_filter()

// checks only C and C++ comments
```

# ZL Implementation Status

- Prototype Compiler
- Most Of C
- Enough C++ to Demonstrate Approach
- Compiled Several Real Applications
  - C: bzip2, gzip
  - C++: randprog
- Compile Time 2-3 slower
- No Impact on Run-time Performance

# Conclusion

## Demonstrated
## ABI Compatibility Through Macros