

AspectC Introduction

Presented by:

Michael Gong and Hans-Arno Jacobsen

CASCON 2006

Crosscutting in C Programs

```
void *NutHeapAlloc(size_t size) {  
    #ifdef NUTDEBUG  
        //code removed  
    #endif  
    NODE **fpp = 0;  
    //code removed  
    #if defined(__arm__) ||  
        defined(__m68k__) ||  
        defined(__H8300H__) || ...  
        while ((size & 0x03) != 0)  
            size++;  
    #endif  
    if (size >= available) {  
        #ifdef NUTDEBUG  
            //code removed  
        #endif  
        return 0;  
    }  
    //code removed  
    ...(next column)
```

```
while (node) {  
    //code removed  
    if (fit) {  
        //split the node if too big  
        if (fit->hn_size > ...) {  
            //code removed  
            *fpp = node;  
        } else  
            *fpp = fit->hn_next;  
    }  
    ...}  
    //code removed  
}  
#ifdef NUTDEBUG  
    //code removed  
#endif  
return fit; }
```

Nut/OS, heap.c

debug concern

system-specific concern

optimization concern

main functionality

Crosscutting in C Programs

```
int is_orphaned_pgrp(int pgrp) {  
    int retval;
```

```
    read_lock(&tasklist_lock);
```

```
    retval =  
        will_become_orphaned  
        _pgrp(pgrp, NULL);
```

```
    read_unlock(&tasklist_lock);
```

```
    return retval;
```

```
}
```

...(next column)

```
int session_of_pgrp(int pgrp) {  
    struct task_struct *p;  
    int sid = -1;
```

```
    read_lock(&tasklist_lock);
```

```
    do_each_task_pid(pgrp,  
        PIDTYPE_PGID, p) {  
        // code removed  
    }
```

```
    // code removed
```

```
    read_unlock(&tasklist_lock);
```

```
    return sid;
```

```
}
```

Linux Kernel 2.6
kernel/exit.c

synchronization concern

main functionality

Crosscutting in C Programs

```
struct lock *  
lock_create(const char *name) {  
    struct lock *lock;  
    lock = kmalloc(sizeof(struct lock));  
    if (lock == NULL) { return NULL; }  
    //code removed  
}
```

synch.c

```
static struct thread *  
thread_create(const char *name) {  
    struct thread *thread =  
        kmalloc(sizeof(struct thread));  
    if (thread == NULL) { return NULL; }  
    //code removed  
}
```

thread.c

```
char * kstrdup(const char *s) {  
    char *z = kmalloc(strlen(s)+1);  
    if (z == NULL) { return NULL; }  
    //code removed  
}
```

OS161

misc.c

```
struct vnode *  
dev_create_vnode(...) {  
    int result;  
    struct vnode *v;  
    v = kmalloc(sizeof(struct vnode));  
    if (v == NULL) { return NULL; }  
    //code removed  
}
```

device.c

error checking concern

main functionality

Crosscutting in C Programs

```
void * rb_find (const struct rb_table* tree, const void *item) {
```

```
    //local variable declaration
```

```
    assert (tree != NULL && item != NULL);
```

```
    //code removed
```

```
}
```

```
void ** rb_probe (struct rb_table *tree, void *item) {
```

```
    //local variable declaration
```

```
    assert (tree != NULL && item != NULL );
```

```
    //code removed
```

```
}
```

```
void * rb_delete (struct rb_table *tree, const void *item) {
```

```
    //local variable declaration
```

```
    assert (tree != NULL && item != NULL);
```

```
    //code removed
```

```
}
```

GNU libavl 2.0.2
rb.c

precondition check concern

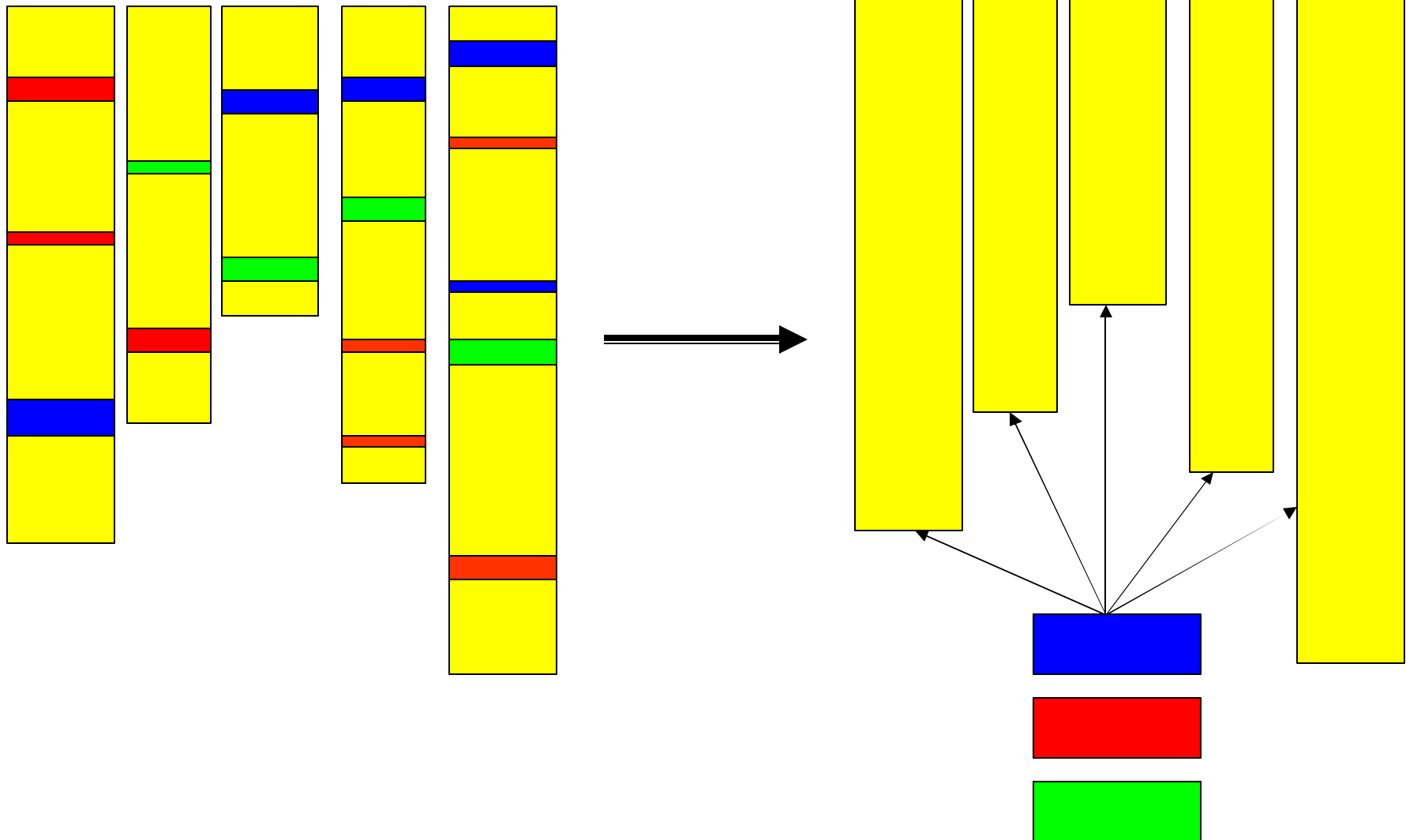
main functionality

Code Summary

- Certain concerns crosscut the principal or core logic
 - a.k.a. *crosscutting concerns*
- Similar concern code scatters across the code
- Different pieces of concern code tangled with core logic
- Scattering, tangling, and crosscutting leads to code
 - that is hard to **read, understand** and **maintain**
 - where the design intent is not cleanly represented
 - where concerns are not well separated and modularized
 - where removing a concern is error-prone

- Crosscutting phenomenon is often *NOT* due to **bad design**
- Related to the characteristics of traditional development techniques
- Decomposition mechanism of traditional development paradigms consists of
 - for C: files, functions, structures
 - for OO: classes, objects, interfaces, methods

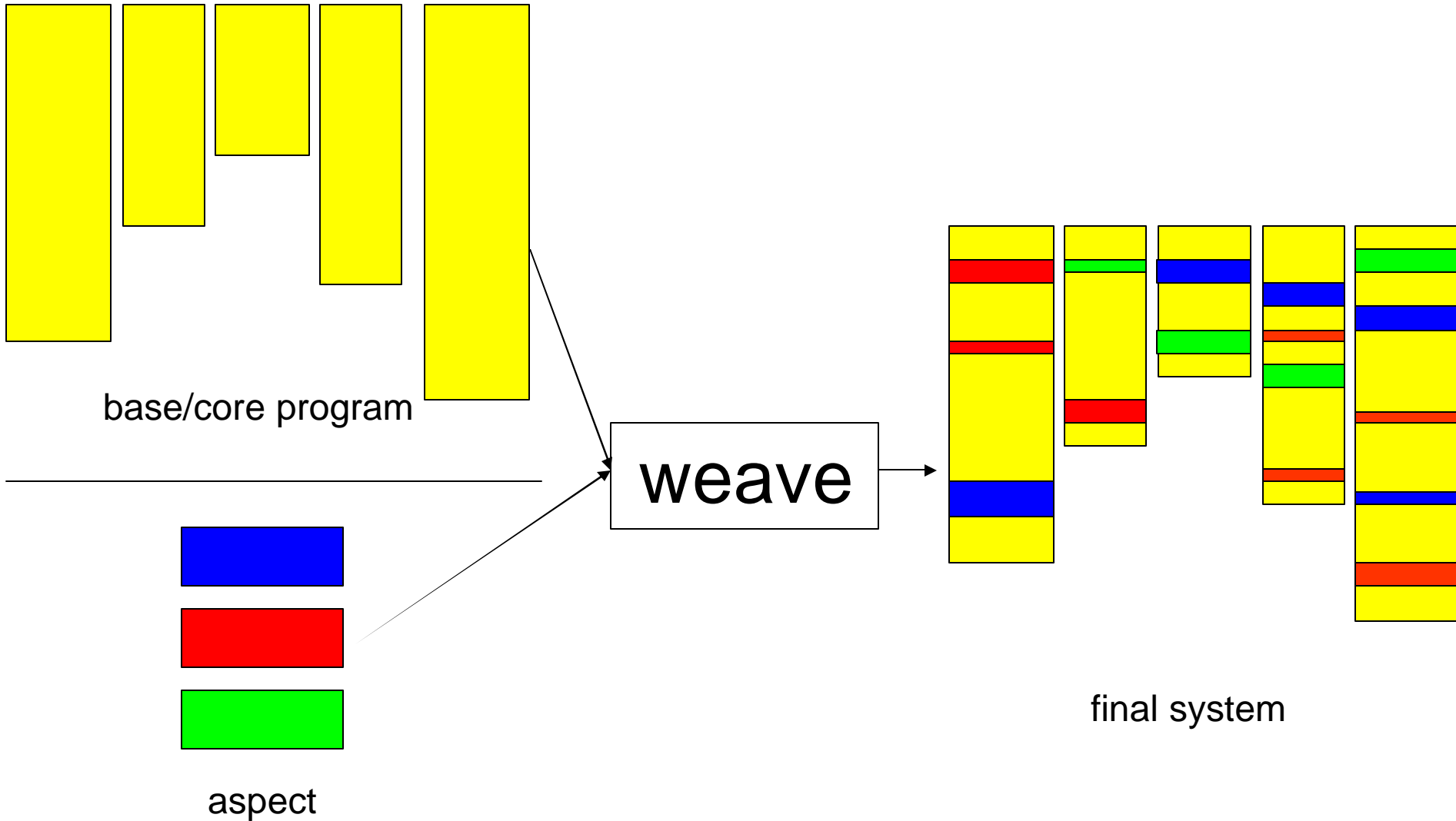
Is There a Solution?



Aspect-oriented Programming

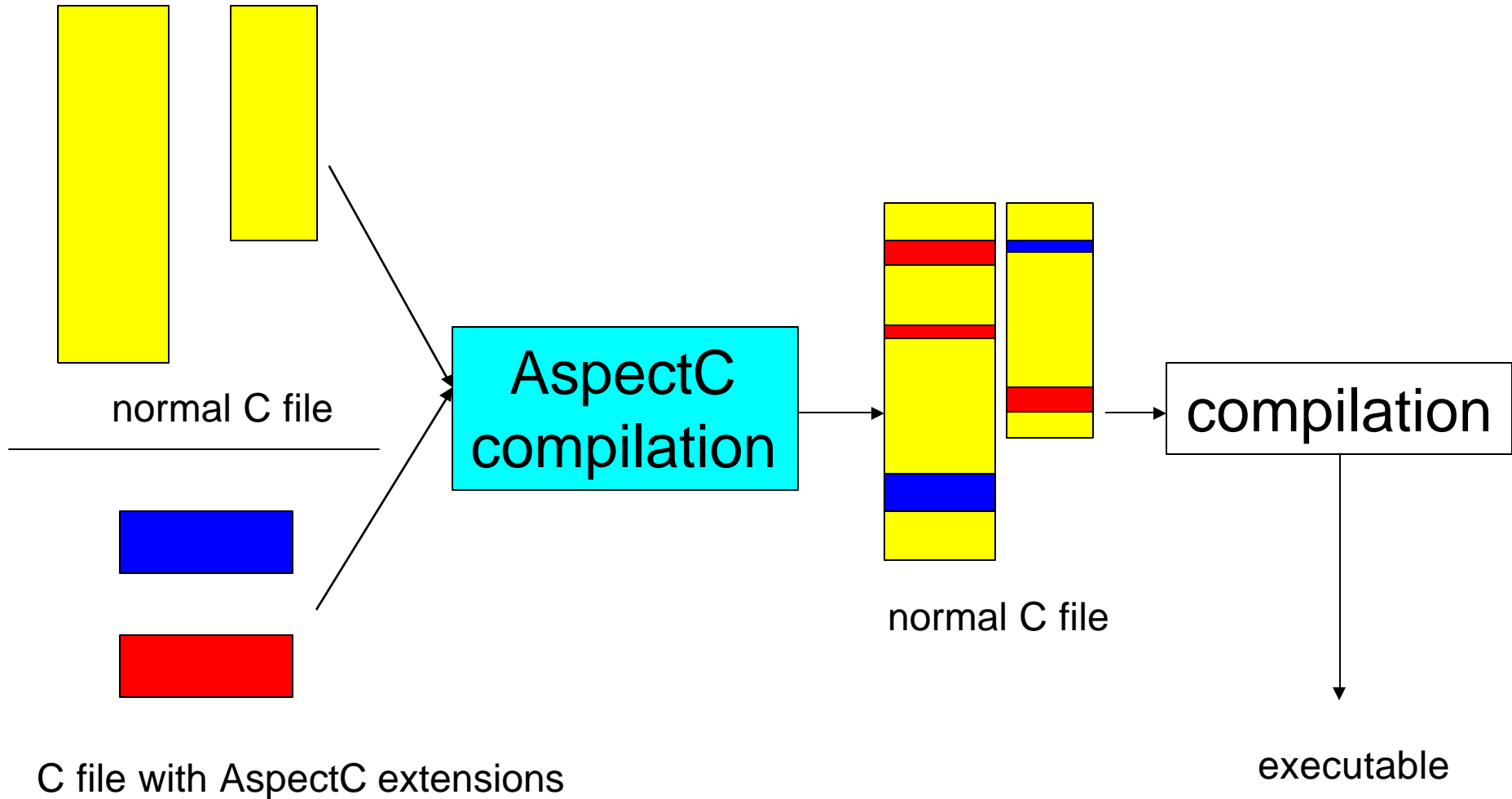
- AOP is
 - a programming paradigm that aims to support the modularization of crosscutting concerns in software
 - **complementary** to existing paradigms
- Emerged about 10 years ago from different research efforts studying the separation of concerns in software
- Supported in industry today by IBM, BEA,...
- AOP support is available for Java, C++, C, PHP, ...
- AspectJ, AspectC++, AspectC, AOPHP, ...

AOP Key Idea



- Designed by Michael Gong and Hans-Arno Jacobsen
 - started around April 2006
 - as an effort by the Middleware Systems Research Group at the University of Toronto
- An aspect-oriented extension to C
- Highlights comprise
 - ANSI-C and C99 compliance
 - gcc source-compatibility
 - Compiler and generated code portability
 - Seamless Linux, Solaris and Windows support (Mac OSX support in progress)
 - Integration into existing build processes possible
 - Code transparency through source-to-source transformations
 - Based on open source license and compiler

AspectC Key Idea



AspectC Features

- **join point**: call and execution join points
- **advice**: before, after and around
- **pointcut**:call(), execution(), args(), infile(), infunc(), result()
- pointcut composition: &&, || , !
- named pointcut
- proceed() call
- wildcard character matching through “\$” and “...”
- recognize gcc extended keywords
 - `__extension__`, `__attribute__`, `__builtin_va_list`, `__inline__` , `__asm`, ...
- **cflow()** support (also under multi-threading)
- reflective information about join points
- static crosscutting support
 - add new struct/union member: `intype()` pointcut and `introduce()` advice
- generated code is thread-safe

AspectC Join Point Model

- join point
 - the **location** in the **base** program where **aspects** take effect

```
void foo (int a) {  
    int x = a;  
    foo2(x);  
}
```

```
void main () {  
    int x , * p;  
    x = 7;  
    p = &x ;  
    foo( *p );  
}
```

function
execution

function call

AspectC Pointcut

- pointcut
 - a language construct to denote join points

execution (void foo(int))

```
void foo (int a) {  
    int x = a;  
    foo2(x);  
}
```

call (void foo2(int))

execution (void main())

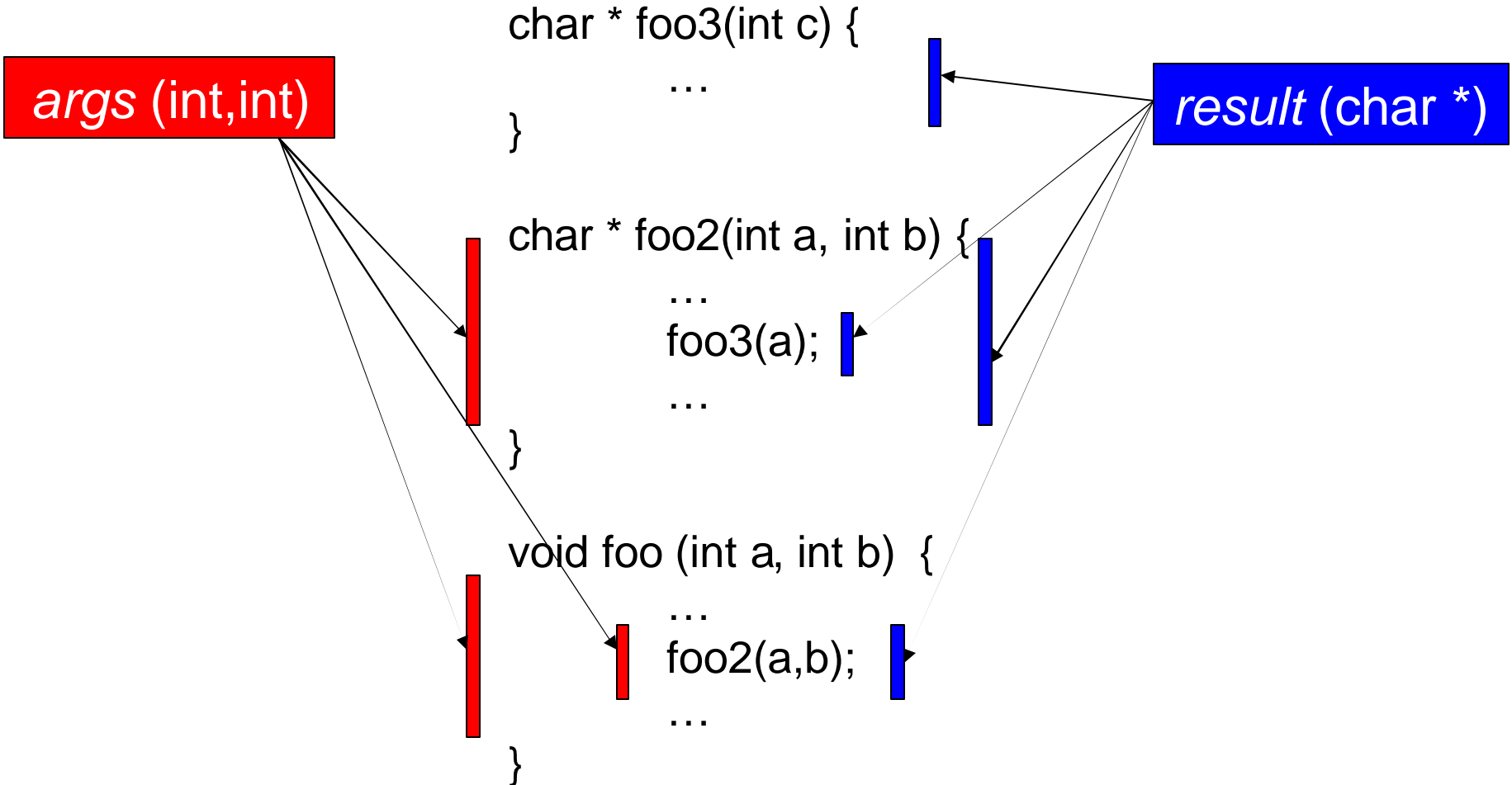
```
void main () {  
    int x , * p;  
    x = 7;  
    p = &x ;  
    foo( *p );  
}
```

call (void foo(int))

AspectC Pointcut

- *args* (parameter-type-list)

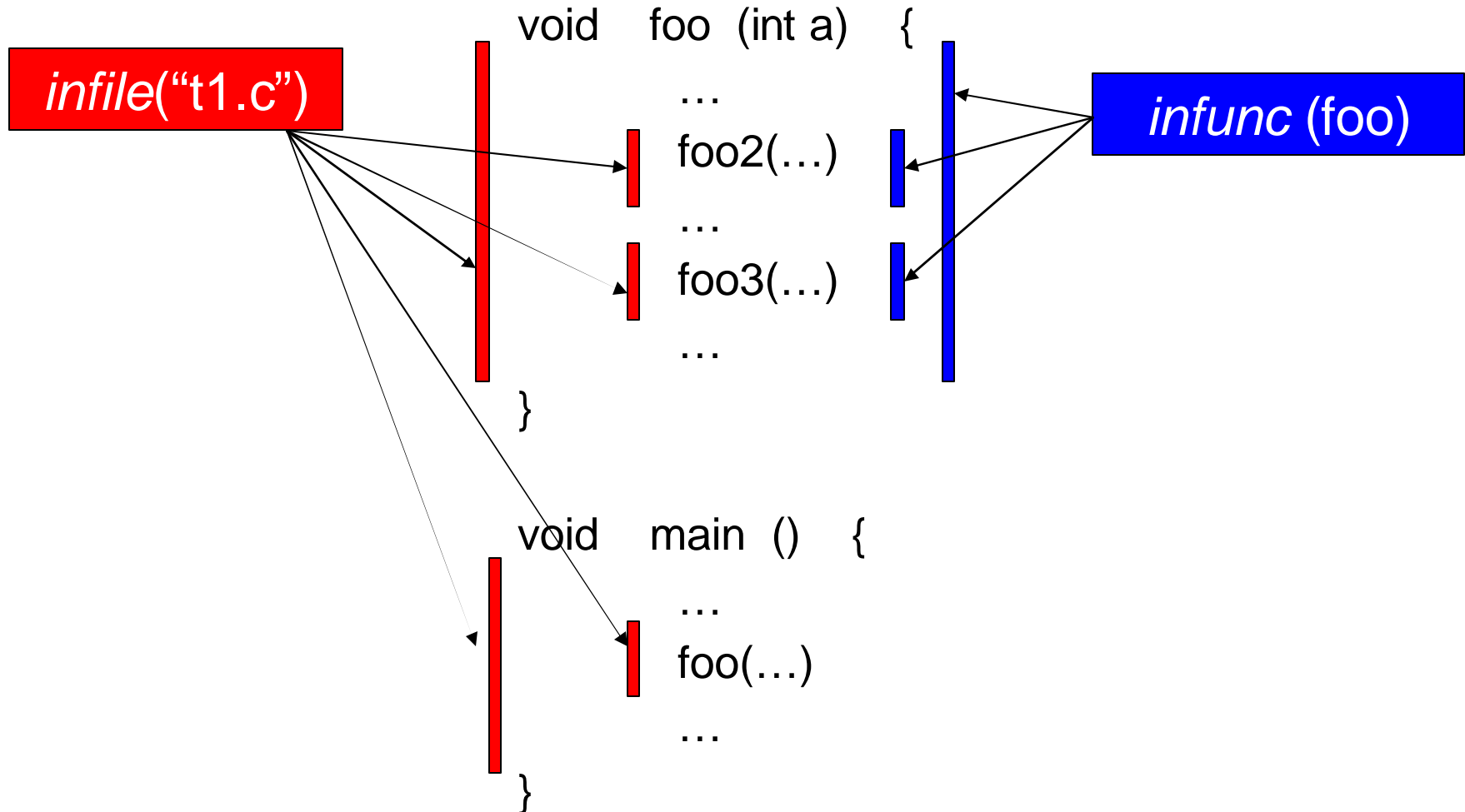
- *result* (return-type)



AspectC Pointcut

□ *infile* ("file-name")

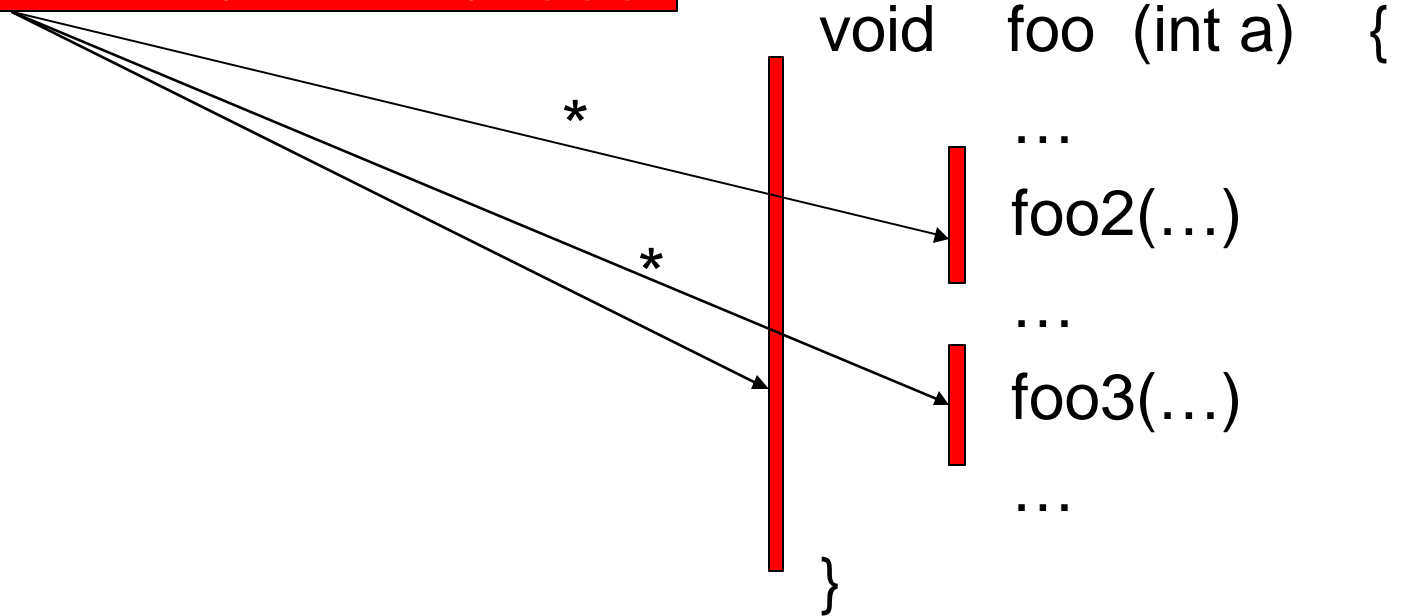
□ *infunc* (func-name)



AspectC Pointcut

- *cflow* (pointcut-declaration)
 - all join points happening under the control flow of other join points

```
cflow ( execution (void foo(int) ) )
```



* all join points happening inside foo2() or foo3() function calls

AspectC Pointcut Composition

- compose pointcuts with `&&`, `||`, `!` (i.e., *and*, *or*, and *not*)

```
infile ("t1.c") && infunc (foo)
```

```
execution (void foo(int)) || args (int,int)
```

```
(result (char *) || infunc (foo)) && ! call (void foo2(int))
```

```
cflow (execution (void foo(int))) && call (void foo2(int))
```

AspectC Named Pointcut

- *pointcut* name (parameter-list) : pointcut-declaration

```
pointcut CallFoo2(): call (void foo2(int))
```

```
CallFoo2() || args (int,int)
```

```
cflow (execution (void foo(int))) && CallFoo2()
```

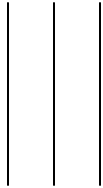
```
(result (char *) || infunc (foo)) && ! CallFoo2()
```

AspectC Pointcut Using Wildcard Character

\$: match any single item

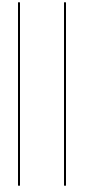
... : match any list of items

```
call (lon$ foo$())
```



```
call (long long foo())  
call (long foo2())  
call (long int foo3())  
...
```

```
args (int , ... , char * )
```



```
args (int , char * )  
args (int, char, char * )  
args (int, int *, char * )  
args (int, char, char , char * )  
...
```

AspectC Advice

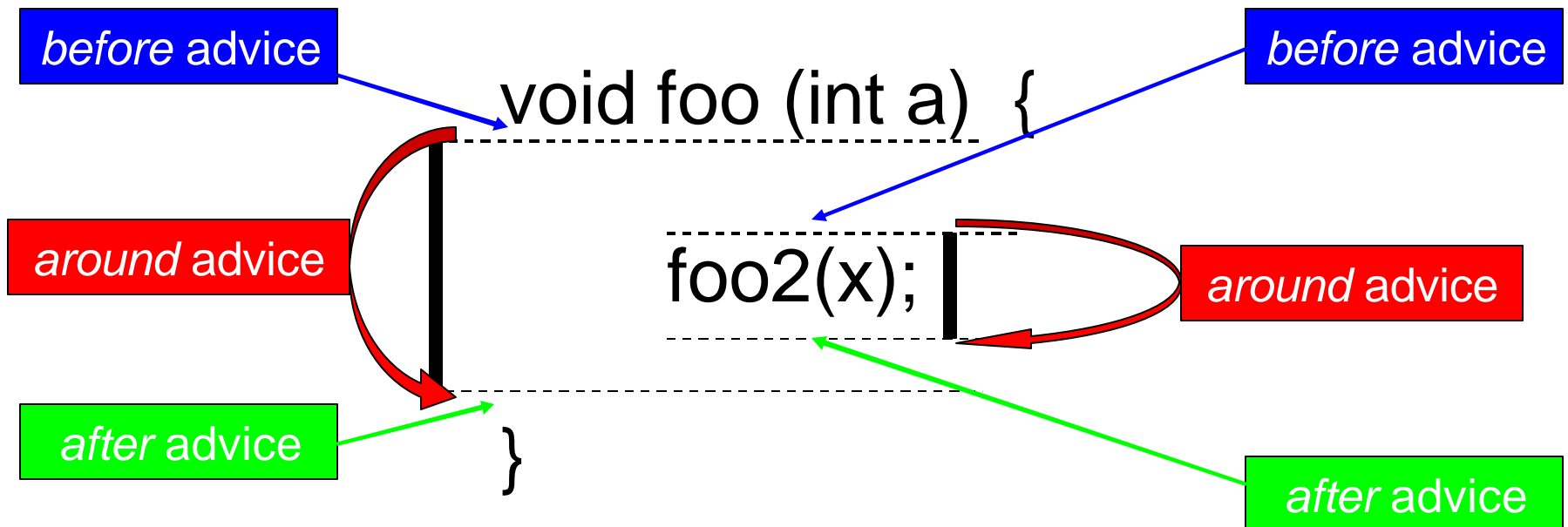
- the code to run for a pointcut

before/after (parameter-list) : pointcut-declaration

{ //advice body }

return-type *around* (parameter-list) : pointcut-declaration

{ //advice body }



AspectC Advice Example

```
before (): execution (void foo(int)) {  
    printf(" before exec\n");  
}
```

```
void foo (int a) {  
    foo2(x);  
}
```

```
void around ():  
    execution (void foo(int)) {  
        printf("around exec\n");  
    }
```

```
after (): execution (void foo(int)) {  
    printf(" after exec\n");  
}
```

AspectC Advice Example

```
before (): call (void foo2(int)) {  
    printf(" before call\n");  
}
```

```
void foo (int a) {  
    -----  
    foo2(x); |  
    -----  
}
```

```
void around ():  
    call (void foo2(int)) {  
        printf("around call\n");  
    }
```

```
after (): call (void foo2(int)) {  
    printf(" after call\n");  
}
```

AspectC Advice Example

- access argument value by using *args* ()
- access return value by using *result* ()

```
before (int i): call (void foo2(int)) && args (i) {  
    printf(" before call foo2, argument = %d\n", i);  
}
```

```
after (int res): call (int foo2(int)) && result (res) {  
    printf(" after call foo2, return %d\n", res);  
}
```

AspectC Advice Example

- around advice
 - invoke original function via *proceed()*

```
void around ( ): call (void foo2(int)) {  
    printf("around begin\n");  
  
    proceed();  
  
    printf("around end\n");  
}
```

```
void foo2(int a) {  
    printf("in foo2\n");  
}  
  
int main() {  
    foo2(3);  
}
```

if no **proceed()** used:
around begin
around end

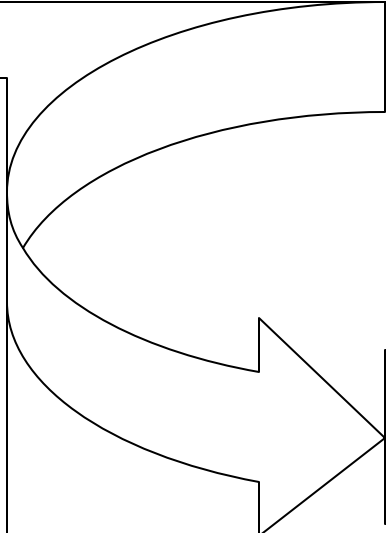
around begin
in foo2
around end

AspectC Advice Example

- reflective information about join point
 - *this->funcName, this->kind*

```
before ( ): call (void foo2(int)) {  
    printf("before %s %s \n", this->kind, this->funcName);  
}
```

```
void foo2(int a) {  
    printf("in foo2\n");  
}  
int main() {  
    foo2(3);  
}
```



```
before call foo2  
in foo2
```

AspectC Advice Example

- wildcard matching increases the usability

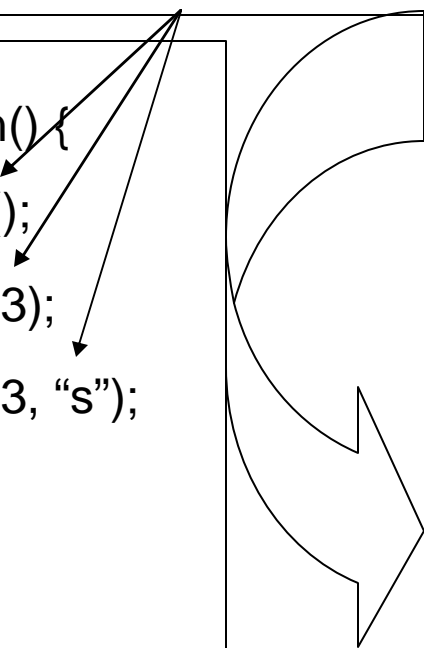
```
before ( ): call (void $(...)) {  
    printf("before %s %s \n", this->kind, this->funcName);  
}
```

```
void fun1() {  
    printf("in fun1\n\n");  
}
```

```
void foo2(int a) {  
    printf("in foo2\n\n");  
}
```

```
void foo3(int a, char * s) {  
    printf("in foo3\n\n");  
}
```

```
int main() {  
    fun1();  
    foo2(3);  
    foo3(3, "s");  
}
```



before call fun1
in fun1

before call foo2
in foo2

before call foo3
in foo3

AspectC Static Crosscutting

- add new data members to struct/union types

```
introduce ( ) : intype ( type-name ) {  
    // new member declaration  
}
```

```
struct X {  
    int a;  
    char b;  
};  
int main() {  
    printf("size of X = %d\n",  
        sizeof(struct X));  
}
```

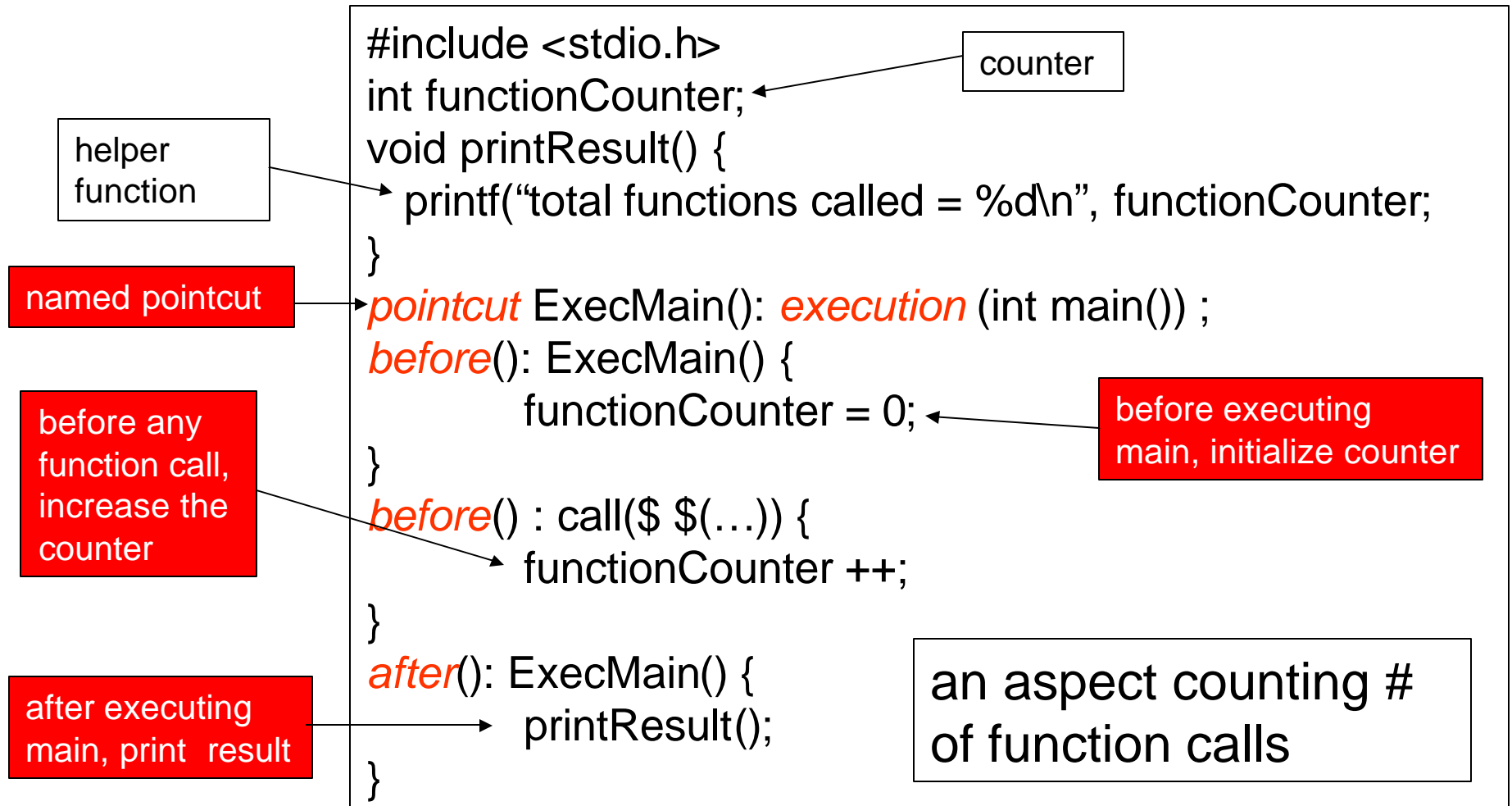
```
introduce ( ): intype ( struct X ) {  
    double x;  
    char * parent;  
}
```

```
struct X {  
    int a;  
    char b;  
    double x;  
    char * parent;  
}
```

size of X = 20

Aspects in AspectC

- Aspect = a file having AspectC extension & C code



Use AspectC in Real C Code

□ GNU libavl

- a collection of binary search trees and balanced tree library routines
- <http://www.stanford.edu/~blp/avl>
- version 2.0.2a
- mostly complete and well-documented
- for simplification, we focus on
 - *red-black tree (RBT) routines and its correctness testing*

Use and Test RBT in Libavl

test.c

```
int main (...) {
  // parse command
  // line options
  // generate insertion and
  // deletion order
  ...
  switch (opts.test) {
    ...
    test_correctness (...)
    ...
  }
  ...
}
```

rb-test.c

```
/* tests tree functions */
Int test_correctness (...) {
  // test insert, delete,
  // copy, etc.
  ... rb_probe (...);
  ...
  ... rb_t_find (...);
  ...
  ... rb_delete (...);
  ...
  ... rb_t_copy (...);
  ...
}
```

rb.c

```
/* function definition */
... rb_probe (...) {
  ...
}
... rb_t_find (...) {
  ...
}
... rb_delete (...) {
  ...
}
... rb_t_copy (...) {
  ...
}
```

Trace Concern

```
int test_correctness (... , int verbosity) { ...
    if (verbosity >= 2) printf (" Inserting %d...\n", insert[i]);
    { void **p = rb_probe (tree, &insert[i]);
      ...
      if (verbosity >= 2)
        printf ("  Checking traversal from item %d...\n",
insert[i]);
      if (rb_t_find (&x, tree, &insert[i]) == NULL) { ... }
      ...
      if (verbosity >= 3)
        printf ("  Deleting item %d.\n", delete[i]);
      deleted = rb_delete (tree, &delete[i]);
      ...
      if (verbosity >= 3)
        printf ("  Re-inserting item %d.\n", delete[i]);
      rb_t_insert (&z, tree, &delete[i])...
      ...
      if (verbosity >= 2)
        printf ("  Deleting %d...\n", delete[i]);
      deleted = rb_delete (tree, &delete[i]);
      ...
      if (verbosity >= 2)
        printf ("  Copying tree and comparing...\n");
      { ...rb_copy (tree, NULL, NULL, NULL); ... } ...}
```

rb-test.c

a **trace concern** controlled by the variable “verbosity”

* it crosscuts the core logic of this function.

* the core logic code is *polluted*.

Trace Aspect

```
#include "rb.h"

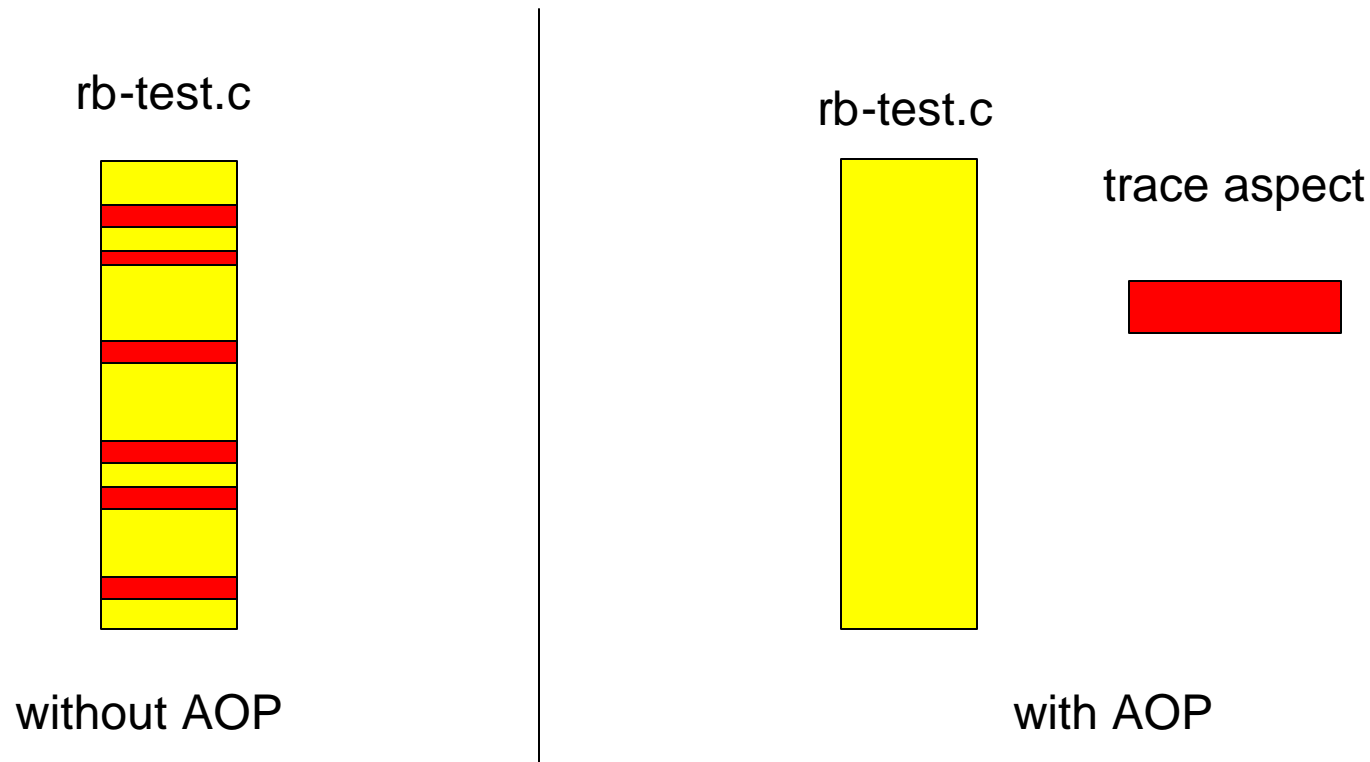
pointcut INTEST (): ifunc(test_correctness);

before(void * node ): INTEST() && call($ rb_probe(...)) && args($, node) {
    printf (" Inserting %d...\n", *(int*)(node));
}
before(void * node) : INTEST() && call($ rb_t_find(...)) && args($, $, node) {
    printf (" Checking traversal from item %d...\n", *(int*)(node));
}
before(const void * node) : INTEST() && call($ rb_delete(...)) && args($, node)
{
    printf (" Deleting item %d.\n", *(int*)(node));
}
before(void * node): INTEST() && call($ rb_t_insert(...)) && args($,$,node){
    printf(" Re-inserting item %d.\n", *(int*)(node));
}
before(): INTEST() && call($ rb_copy(...)) {
    printf (" Copying tree and comparing...\n");
}
}
```

Trace Aspect

□ Benefits

- code is modularized in 1 file
- can be easily plugged / unplugged from the core program



Node Count Concern

rb.h

```
struct rb_table {
    struct rb_node *rb_root;
    ...
    /* node count */
    size_t rb_count;
    ...
};
```

Keeping a node count is **not** required for the operation of BST tree, so it is an optional feature

→ a crosscutting concern

```
struct rb_table * rb_create (...) { ...
    tree->rb_count = 0; ...
}
void ** rb_probe (struct rb_table *tree, ...) { ...
    tree->rb_count++; ...
    ...
}
void * rb_delete (struct rb_table *tree,...) { ...
    tree->rb_count --; ...
}

struct rb_table * rb_copy (const struct rb_table
*org, ...) {...
    new->rb_count = org->rb_count;
    if (new->rb_count == 0)
        return new;
    ...
}
```

rb.c

Node Count Aspect

```
#include <stdlib.h>
```

```
#include "rb.h"
```

```
introduce():
```

```
  intype (struct rb_table) {  
    size_t rb_count;  
  }
```

introduce the
count member

```
after(struct rb_table * newtable) :
```

```
  execution($ rb_create(...)) && result (newtable)  
{  newtable->rb_count = 0; }
```

modify the count

```
after(void ** res, struct rb_table * tree) :
```

```
  call($ rb_probe(...)) && result(res) && args (tree, $)  
{  if(res) tree->rb_count ++; }
```

```
after(void * res, struct rb_table * tree):
```

```
  call($ rb_delete(...)) && result (res) && args (tree, $)  
{  if(res) tree->rb_count --; }
```

```
before(const struct rb_table * tree) :
```

```
  call($ print_whole_tree(...)) && args(tree, $)  
{  printf ("rbcount = %d , ", tree->rb_count); }
```

```
struct rb_table *
```

```
around(const struct rb_table * org,  
        struct libavl_allocator *allocator) :  
  call ($ rb_copy(...)) && args (org, ..., allocator)
```

```
{  
  if(org->rb_count == 0) {  
    return rb_create(org->rb_compare,  
                    org->rb_param,  
                    allocator != NULL ? allocator  
                        org->rb_alloc);  
  } else {  
    return proceed();  
  }  
}
```

```
after(struct rb_table * new, struct rb_table * org) :  
  call($ rb_copy(...)) && result(new) && args(org,  
  ...)  
{  
  if(new) new->rb_count = org->rb_count;  
}
```

print count value for verification

Memory Profiling Concern

- requirement:
 - need to know how much memory is allocated to test RBT
- a memory profiling concern
 - a classical crosscutting concern
 - it is not required for the core logic of RBT operations or testing

Memory Profiling Aspect

```
#include <stdlib.h>
```

```
size_t totalMemoryMalloced ;
```

```
after(size_t mem): call($ malloc(...)) && args(mem) {  
    totalMemoryMalloced += mem;  
}
```

Hope you could figure out
how it works by now 😊

```
before(): execution($ main(...)) {  
    totalMemoryMalloced = 0;  
}
```

Unfortunately, it does not
give the correct answer 😞

```
after(): execution($ main(...)) {  
    printf("aspect: total memory allocated = %d\n",  
totalMemoryMalloced);  
}
```

Why Not Work and Solution

test.c

malloc() is called

```
int main (...) {  
    ...  
    switch (opts.test) {  
        ...  
        test_correctness (...) {  
            ...  
        }  
    }  
}
```

We should only record all memory allocated inside the `test_correctness()`

```
after(size_t mem): call($ malloc(...)) && args(mem) {  
    totalMemoryMalloced += mem;  
}
```

! solution !

cflow()

```
after(size_t mem): call($ malloc(...)) && args(mem)  
    && cflow (call($ test_correctness(...))) {  
    totalMemoryMalloced += mem;  
}
```

Using the AspectC Compiler in Make

- ❑ core/aspect files should be preprocessed
- ❑ suffix for core file : .mc (*)
- ❑ suffix for aspect file: .ac (*)

trace: test.c rb.c rb-test.c trace.ac preprocess

gcc -E test.c > test_mc.mc

gcc -E rb.c > rb_mc.mc

gcc -E rb-test.c > rb-test_mc.mc

cp trace.ac trace_temp.c

gcc -E trace_temp.c > trace_temp.ac

acc test_mc.mc rb_mc.mc rb-test_mc.mc trace_temp.ac

gcc -o rbtest_aspect -g test_mc.c rb_mc.c rb-test_mc.c trace_temp.c

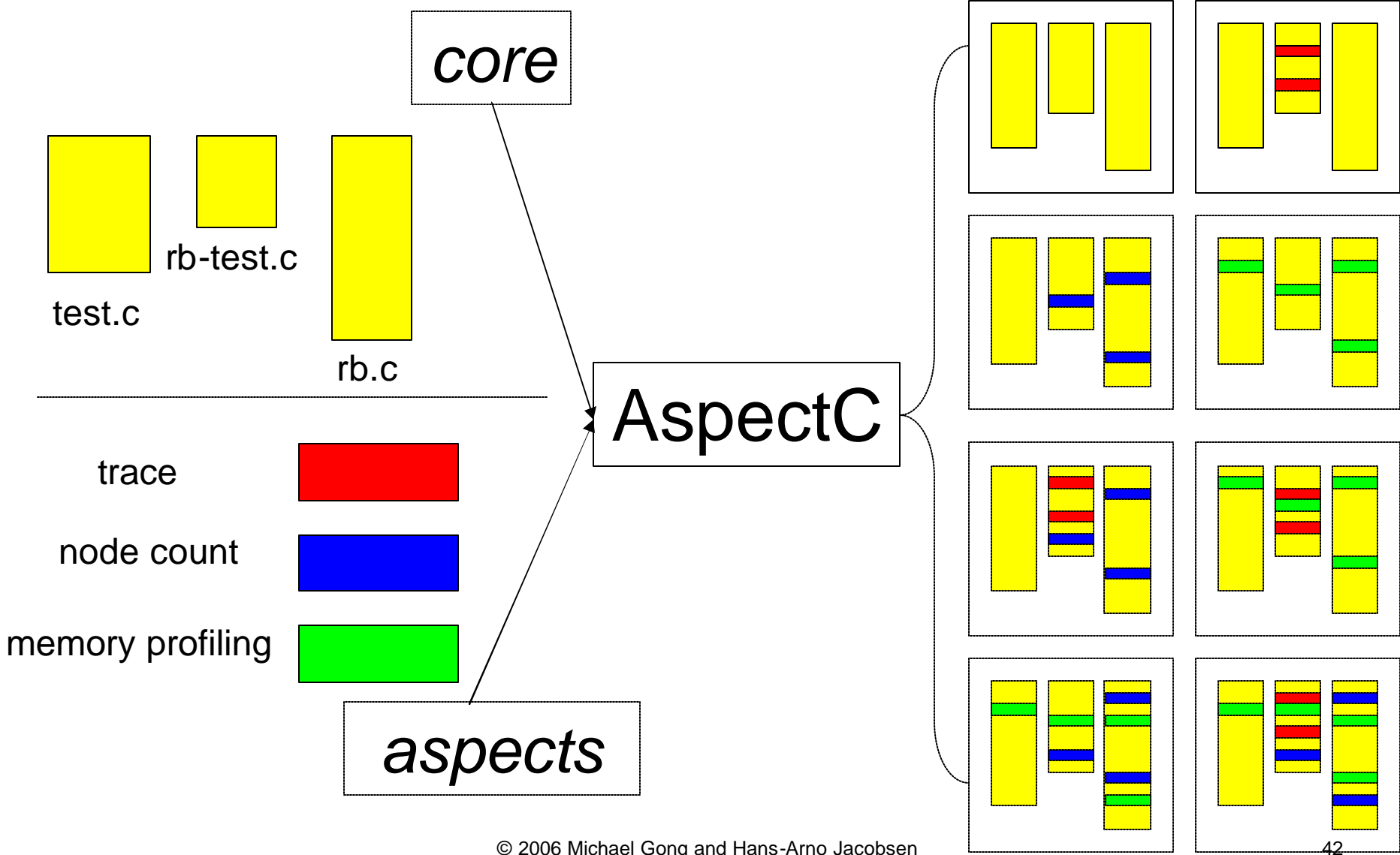
Unfortunately, gcc doesn't recognize ".ac" suffix, we have to copy it to a ".c" file ☹

AspectC compiler !

generated by AspectC compiler.

* file suffix rule might be revised in the future.

AOP-based Software Product Lines



□ AspectC Compiler

- semantic checking
- debugging on the original source file
- global static crosscutting
 - introduce function/variable/header files in file scope

□ Case Studies

- thread-RBT and RBT with parent pointer in Libavl
 - use aspects because they are crosscutting concerns
- refactor an embedded operating system: EtherNUT
- refactor the C-based Orbit object request broker

Related Work

- Gregor Kiczales, *et al.*
 - Aspect-Oriented Programming. ECOOP 1997.
 - THE paper introducing AOP
- Yvonne Coady, *et al.*
 - Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, *FSE 2001*.
 - First research project on applying AOP to C

Related Work

- Some AspectC language design is inspired by
 - AspectJ (www.eclipse.org/aspectj)
 - most mature and widely-used AOP language and tool
 - AspectC++ (www.aspectc.org)
 - most mature application of AOP to C++
 - it could handle plain C code by generating C++ code ☹
 - CrossCutting C Compiler (C4 toolkit)
 - <http://c4.cs.princeton.edu/>
 - aims to introduce AOP to C by “*observing*” the developer
 - implemented in Java

More Information

□ AOP

- <http://www.aosd.net>

Thanks for
reading! 😊

□ AOP on C

- http://www.aosd.net/wiki/index.php?title=Aspects_in_C

□ AspectC

- <http://www.AspectC.net>
- latest version is 0.3 (October, 2006)
- source code, compiler manual, examples, tutorials, and much more
- *we welcome any comments or feedback*