Write gcc in C++

Ian Lance Taylor
Google

June 17, 2008
C++

- C++ is a standardized, well known, popular language.
- C++ is nearly a superset of C90 used in gcc.
- The C subset of C++ is just as efficient as C.
- C++ supports cleaner code in several significant cases.
- C++ makes it easier to write cleaner interfaces by making it harder to break interface boundaries.
- C++ never requires uglier code.
- C++ is not a panacea but it is an improvement.
VEC or vector?

/* C */
typedef struct loop *loop_p;
DEF_VEC_P (loop_p);
DEF_VEC_ALLOC_P (loop_p, gc);

VEC (loop_p, gc) *superloops;
VEC_reserve (loop_p, gc, superloops, depth);
VEC_index (loop_p, superloops, depth)
VEC_quick_push (loop_p, superloops, father);
VEC or vector?

/* C */
typedef struct loop *loop_p;
DEF_VEC_P (loop_p);
DEF_VEC_ALLOC_P (loop_p, gc);

VEC (loop_p, gc) *superloops;
VEC_reserve (loop_p, gc, superloops, depth);
VEC_index (loop_p, superloops, depth)
VEC_quick_push (loop_p, superloops, father);

// C++
typedef std::vector<struct loop*, gc_allocator> loop_vec;
loop_vec* superloops;
superloops->reserve(depth);
superloops[depth];
superloops->push_back(father);
# Write gcc in C++

Ian Lance Taylor
Google

```c
/* C */
	tree_contains_struct [VAR_DECL][TSDECL_WITH_VIS] = 1;

#define CONTAINS_STRUCT_CHECK(T, STRUCT) __extension__
	({
		__typeof__ (T) __t = (T); 
		if (tree_contains_struct[TREE_CODE(__t)][(STRUCT)] != 1)
			tree_contains_struct_check_failed (__t, (STRUCT), __FILE__,
				__LINE__, __FUNCTION__); 
		__t; })

#define DECL_WITH_VIS_CHECK(T) CONTAINS_STRUCT_CHECK (T, TSDECL_WITH_VIS)
#define DECL_DEFER_OUTPUT(NODE) \ 
	(DECL_WITH_VIS_CHECK (NODE)->decl_with_vis.defer_output)

struct tree_decl_with_vis GTY(())
{
	struct tree_decl_with_rtl common;
... 
	unsigned defer_output:1;
};

struct tree_var_decl GTY(())
{
	struct tree_decl_with_vis common;
};
```
tree_contains_struct

/* C */

tree_contains_struct [VAR DECL] [TS DECL WITH VIS] = 1;
#define CONTAINS_STRUCT_CHECK(T, STRUCT) __extension__
({
    _typeof (T) const __t = (T);
    if (tree_contains_struct [TREE_CODE(__t)][(STRUCT)] != 1)
        tree_contains_struct_check_failed (__t, (STRUCT), __FILE__,
        __LINE__, __FUNCTION__);
    __t; })
#define DECL_WITH_VIS_CHECK(T) CONTAINS_STRUCT_CHECK (T, TS DECL WITH VIS)
#define DECL_DEFER_OUTPUT(NODE) \
    (DECL_WITH_VIS_CHECK (NODE) → decl_with_vis.defer_output)
struct tree_decl_with_vis GTY() {
    struct tree_decl_with_rtl common;
    ...
    unsigned defer_output:1;
};
struct tree_var_decl GTY() {
    struct tree_decl_with_vis common;
};

// C++
template <T> T* check_non_null(T* p) { gcc_assert (p); return p; }
#define IS_STRUCT_CHECK(T, STRUCT) (check_non_null(dynamic_cast<T*>(STRUCT)))
#define DECL_WITH_VIS_CHECK(T) IS_STRUCT_CHECK (T, tree_decl_with_vis)
#define DECL_DEFER_OUTPUT(NODE) \
    (DECL_WITH_VIS_CHECK (NODE) → decl_with_vis.defer_output)
class tree_decl_with_vis : public tree_decl_with_rtl {
    ...
    unsigned defer_output:1;
};
class tree_var_decl : public tree_decl_with_vis {
};
TARGET or Target?

```c
/* C */
/* target.h */
    void (* init_builtins) (void);
/* targhooks.h */
#define TARGET_INIT_BUILTINS hook_void_void
/* i386.c */
#undef TARGET_INIT_BUILTINS
#define TARGET_INIT_BUILTINS ix86_init_builtins
static void
ix86_init_builtins (void)
{
    ...
}
```
TARGET or Target?

```c
/* C */
/* target.h */
    void (*init_builtins) (void);
/* targhooks.h */
#define TARGET_INIT_BUILTINS hook_void_void
/* i386.c */
#undef TARGET_INIT_BUILTINS
#define TARGET_INIT_BUILTINS ix86_init_builtins
static void
ix86_init_builtins (void)
{
    ...
}

// C++
// target.h
class Target
{
    virtual void init_builtins () { }
};
// i386.c
class Target_i386 : public class Target
{
    void
    init_builtins ()
    {
        ...
    }
};
```
htab or unordered_map?

/* C */

htab_t exits;

return htab_find_with_hash (exits, e, htab_hash_pointer (e));

slot = htab_find_slot_with_hash (exits, e,
                                 htab_hash_pointer (e),
                                 add ? INSERT : NO_INSERT);

if (slot)
{
    if (add)
        *slot = add;
    else
        htab_clear_slot (exits, slot);
}
htab or unordered_map?

```c
/* C */
htab_t exits;

return htab_find_with_hash (exits, e, htab_hash_pointer (e));

slot = htab_find_slot_with_hash (exits, e,
    htab_hash_pointer (e),
    add ? INSERT : NO_INSERT);

if (slot)
{
    if (add)
        *slot = add;
    else
        htab_clear_slot (exits, slot);
}

// C++
typedef std::tr1::unordered_map<edge, struct loop_exit*> exit_map;
exit_map exits;

exit_map::iterator p = exits.find(e);
return p != exits.end() ? NULL : p->second;

if (add)
    exits[e] = add;
else
    exits.erase(e);
```
Garbage collection or smart pointers?

- GCC generates temporary garbage which is only freed by `ggc_collect`.
  - `ggc_collect` is expensive—scales by total memory usage.
- C++ permits reference counting smart pointers.
  - Fast allocation.
  - Lower total memory usage.
  - Copying a pointer adds an increment instruction.
  - Letting a pointer go out of scope adds a decrement and a test.
  - Reference counts are normally in memory cache, unlike `ggc_collect`.
- We may want to use a mixture of reference counting and garbage collection.
Why not C++?

- C++ is too slow!
- C++ is too complicated!
- C++ library is a bootstrap problem!
- The FSF doesn’t like it!
Why not C++?

- C++ is too slow!
  - C++ is only slower when using optional features which aren’t in C.
  - Sometimes C++ is faster (e.g., STL functions).
  - We would only use features which are worthwhile.
- C++ is too complicated!

- C++ library is a bootstrap problem!

- The FSF doesn’t like it!
Why not C++?

- C++ is too slow!
  - C++ is only slower when using optional features which aren’t in C.
  - Sometimes C++ is faster (e.g., STL functions).
  - We would only use features which are worthwhile.

- C++ is too complicated!
  - It’s just another computer language.
  - Maintainers will ensure that gcc continues to be maintainable.

- C++ library is a bootstrap problem!

- The FSF doesn’t like it!
Why not C++?

- C++ is too slow!
  - C++ is only slower when using optional features which aren’t in C.
  - Sometimes C++ is faster (e.g., STL functions).
  - We would only use features which are worthwhile.

- C++ is too complicated!
  - It’s just another computer language.
  - Maintainers will ensure that gcc continues to be maintainable.

- C++ library is a bootstrap problem!
  - C++ compilers are widely available, including older versions of gcc.
  - We would have to ensure that gcc version N - 1 could always build gcc version N.
  - We will link statically against libstdc++.

- The FSF doesn’t like it!
Why not C++?

- C++ is too slow!
  - C++ is only slower when using optional features which aren’t in C.
  - Sometimes C++ is faster (e.g., STL functions).
  - We would only use features which are worthwhile.

- C++ is too complicated!
  - It’s just another computer language.
  - Maintainers will ensure that gcc continues to be maintainable.

- C++ library is a bootstrap problem!
  - C++ compilers are widely available, including older versions of gcc.
  - We would have to ensure that gcc version N - 1 could always build gcc version N.
  - We will link statically against libstdc++.

- The FSF doesn’t like it!
  - The FSF is not writing the code.
Proposal

- Permitting C++ in gcc will require steering committee approval.
- I plan to create a gcc-in-c++ branch for people to experiment with building gcc in C++.
  - The interaction of garbage collection and STL constructs will need to be resolved.