Advanced C Programming

Declarations, External Names, Memory Layout

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

04.11.2008

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Overview

Literature

- Harbison & Steele, C — A Reference Manual, Chapter 4
- ISO/IEC 9899:1999, Chapter 6
- C Design Rationale, Chapter 6.2.2

See course website for links

Declarations

- Associate identifier with C object
- C objects:
  - variable
  - function
  - type
  - type tag
  - structure and union components
  - enum constants
  - labels (for `goto`)
  - preprocessor macros

Structure of Declarations

A declaration in C consists of

- Storage class specifier: `extern`, `static`, `auto`, `register`
  - For syntactical reasons, `typedef` is also a storage class specifier
- Type qualifiers: `const`, `volatile`, `restrict` (C99)
  - redundant occurrences are error in C89 but not in C99!
- Type specifiers: `unsigned`, `signed`, `char`, `int`, ...
  - C89: missing type specifier equals to `int`
- Declarator
  - Can be left out in certain cases
  - considered bad style, so we don't elaborate on it
- Initializer (One or none)

Example

```
unsigned volatile long extern int const j;
extern const volatile unsigned long int i = 3;
```

Convention

Use following order: storage class, qualifier, specifier

Attributes of Declarations

Each declaration defines several attributes of the declared object:

- **Scope** Range in the program text where the object's identifier is declared
- **Visibility** Range in the program text where the declared object can be accessed with its identifier
- **Name Space** Which kinds of objects must have distinct names if they shall be referenced at the same time in the same scope
- **Extent** The lifetime of the object during program runtime
- **Linkage** Is the object visible from other translation units?

Visibility

- One declaration can hide another

```
int foo = 10;
int main(void) {
    float foo; /* this foo hides outer foo */
    ...
}
```

- Where does the hiding start?

```
{
    int i = 0;
    {
        int j = i;
        int i = 10;
    }
}
```

`j == 0` or `j == 10`?

Visibility

- One declaration can hide another

```
int foo = 10;
int main(void) {
    float foo; /* this foo hides outer foo */
    ...
}
```

- Where does the hiding start?

```
{
    int i = 0;
    {
        int j = i;
        int i = 10;
    }
}
```

`j == 0` or `j == 10`?

Rule

Scope starts at declaration point, not at start of enclosing block

Name Spaces

- The same identifier can declare different kinds of objects at the same time (aka overloading)
- These object have to be in different name spaces (overloading classes)
- C defines the following
  - Proprocessor macro names
  - `goto` labels
  - `struct`, `union`, and `enum` tags
  - Names of components of `struct`s and `union`s
  - The rest: variables, functions, `typedef` names

Name Spaces

- The same identifier can declare different kinds of objects at the same time (aka overloading)
- These object have to be in different name spaces (overloading classes)
- C defines the following
  - Proprocessor macro names
  - `goto` labels
  - `struct`, `union`, and `enum` tags
  - Names of components of `struct`s and `union`s
  - The rest: variables, functions, `typedef` names
- Example

```
extern int howmany;          /* rest name space */
extern char str[10];         /* rest */
typedef double howmany(); /* rest: conflict! */
extern struct str { int a, b; } x; /* tag: no conflict */
```

Extent

- Lifetime of object at runtime
- Static extent
  - Storage allocated before program start
  - Storage remains allocated until program ends
  - All functions, top-level declared variables, and local variables declared `static` or `extern` have static extent
- Local extent
  - Created on entry to a block or function
  - Destroyed at block's (function's) exit
  - Re-created each time block/function is entered

Storage Classes

auto and register

auto (local variables)

- Cannot be used for global variables
- Seldom used explicitly
- Will have a revival in the new C++0x standard

register (local variables and function parameters)

- Equivalent to auto but:
- Hint for the compiler that variable is used frequently
- Nowadays, rarely used
- Modern register allocation is powerful enough

Storage Classes

`extern` and `static`

`extern`

- Static extent
- External Linkage
- Variables: non-defining declaration:
  no memory will be allocated for the variable
- Functions: Default for top-level defined functions

`static`

- Static extent
- Internal linkage
- Variables: tentative declaration:
  If no initializer is given, then variable will be initialized to 0

Attention!

Note that top-level defined variables without storage class are not `extern`. They have external linkage, but that is not identical to `extern`

Type Qualifiers

const

- Helps you:
  avoid unintentional write to data that should not be written to
- Helps the compiler:
  Can optimize memory access because it knows that `const` variables cannot be modified
- Pay attention to pointer rules:

```
int * const const_pointer;
const int *pointer_to_const;
```

- Never cast `const` variables to non-`const` ones
  ☞ write access leads to undefined behavior
- Example

```
int *p, i;
const int *pc, ic;
pc  = p = &i;       /* ok */     pc  = &ic;  /* ok */
*p  = 5;            /* ok */     *pc = 5;    /* invalid */
p   = &ic;          /* invalid */
p   = pc;           /* invalid */
p   = (int *) &ic;  /* works, but dangerous */
```

Type Qualifiers

const — Usage Example

- Use const for getters

```
struct coord {
    int x, y;
}

int coord_set_x(struct coord *c, int x) {
c->x = x;
}

int coord_get_x(const struct coord *c) {
    return c->x;
}
```

Type Qualifiers

const — Usage Example

- Use const for getters

```
struct coord {
    int x, y;
}

int coord_set_x(struct coord *c, int x) {
c->x = x;
}

int coord_get_x(const struct coord *c) {
    return c->x;
}
```

Rules

- Understand usage of const
- Use const where ever possible
- Never de-const-ify code
- Never cast const pointer to non-const pointer

Type Qualifiers

`volatile`

- Important for concurrency (software and hardware!)
- Usually, the compiler has some freedom where to store the contents of variables
- Dependent on this storage location, concurrent updates might be seen or not
- Example

```
int flag;
void foo(void) {
    if (flag)
        do_something;

   /* flag modified
  by another thread */

    if (flag)
        do_another_thing;
}
```

- Flag modified by another thread between two accesses
- Assume compiler keeps `flag` in register in `foo`
- Reasonable optimization to save memory accesses
- Concurrent update invisible!
- When should contents of a variable be visible to other threads?

Type Qualifiers

`volatile` — Sequence Points

- When do the effects on `volatile` variables need to be visible?
- C standard defines so-called sequence points
- Between those sequence points `volatile` variables are not synchronized with memory
- Basically, after each statement
- But not within (non-short-circuit) expressions
- Another argument to not have side effects in expressions
- See Annex C of C standard

Type Qualifiers

`restrict` (C99)

- Can only be used with pointers
- Annotation to help the compiler
- Helps memory disambiguation (later in the course)
- Example:

```
void add(int n, int * restrict a, int * restrict b) {
    int i;

    for (i = 0; i < n; i++)
        a[i] += b[i];
}
```

- Inside `add`, the compiler can assume that arrays `a` and `b` do not overlap
- If they do, the behavior may be undefined

Declarators

Overview

- C declarators can be hard to read:

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

- Rationale: Look like the use of the declared variable
- 2 golden rules:
  1. Go from inner to outer
  2. Arrays and functions have higher priority that pointers
- Example:

```c
int (*x)[5]; /* Pointer to an array of 5 ints */
int *x[5];   /* Array of five pointers to ints */
```

- More examples:

```c
int *(*(*fp1)(int))[10];
float (*(*b())[])();
void *(*c)(char, int (*)());
...
```

Declarators

- Do not use complicated declarators
- Use typedefs to break them into pieces

```
const char *(*(*x)[10])(void *);
```

- Is a pointer to an array of pointers to functions, which take a void pointer and return a string
- Write:

```
typedef const char *(*printer_t)(void *);
typedef printer_t printers_t[10];
printers_t *x;
```

- You must be able to read array and function pointer typedefs

Initializations

Guidelines

- Separate declaration and initialization
- Multiple, comma-separated initializations are hard to read
- Avoid visibility problems (see earlier slides)
- Do not initialize eagerly

Not good

```
int x = 0;
/* x not used here */
x = y + 1;
```

Good

```
int x;
/* x not used here */
x = y + 1;
```

- Compiler (with `-Wall`) will tell you if variable is potentially undefined
- Limit scope as much as possible:

Not good

```
int x = 0;
if (...) {
    x = f();
    ...
    printf("%d", x);
}
```

Good

```
if (...) {
    int x;
    x = f();
    ...
    printf("%d", x);
}
```

Implicit Declarations

- Usually, all identifiers have to be declared before they are used
- In C89 there is one exception that can lead to hard-to-find bugs

```
void f(void) {
    g(2.718);
}

void g(int x) {
    printf("%d\n", x);
}
```

- Will print garbage: depending on endianess, the lower or higher 32 bits of the double `2.718`
- If function prototype not given before call

```
int func();
```

  is assumed
- Prototype does not describe the function but how it is called!
- Thus: Always provide correct prototype

External Names

- How to make objects visible/hidden to other translation units?
- Easy for functions:
  - Give `static` for local linkage
  - Give or omit `extern` for external linkage
  - Whole program needs exactly one definition for a (used!) function in one of the translation units
- More complicated for variables:
  - `static` imposes local linkage
  - Else we have external linkage
  - Giving `extern` or not makes a difference!
  - Remember: External linkage does not require `extern`

Major Question

Which declaration of a global-linkage variable creates storage?
☞ There are four models (!) and the standard

External Names

Linkage Models

Common

- All declarations with external linkage (no matter if `extern` or not) create storage.
- The linker puts all definitions of the same name to the same address
- Named after `FORTRAN` common zones

Relaxed Ref/Def

- Declarations with `extern` are pure references
  ☞ no storage allocated
- Definitions are declarations without storage class
- In all translation units, at least one definition must exist
- Referencing declarations of unused vars may be ignored

Strict Ref/Def

- Like relaxed Ref/Def, but exactly one definition must exist

Initialization

- Only declarations that initialize the variable create storage

Linkage Models

Overview (from C99 Design Rationale, Chapter 6.2.2)

<table>
<tr><th>Model</th><th>File 1</th><th>File 2</th></tr>
<tr><td>Common</td><td><pre>extern int i;
int main( )
{
    i = 1;
    second( );
}</pre></td><td><pre>extern int i;
void second( )
{
    third(i);
}</pre></td></tr>
<tr><td>Relaxed Ref/Def</td><td><pre>int i;
int main( )
{
    i = 1;
    second( );
}</pre></td><td><pre>int i;
void second( )
{
    third(i);
}</pre></td></tr>
<tr><td>Strict Ref/Def</td><td><pre>int i;
int main( )
{
    i = 1;
    second( );
}</pre></td><td><pre>extern int i;
void second( )
{
    third(i);
}</pre></td></tr>
<tr><td>Initializer</td><td><pre>int i = 0;
int main( )
{
    i = 1;
    second( );
}</pre></td><td><pre>int i;
void second( )
{
    third(i);
}</pre></td></tr>
</table>

Linkage Models

The Standard

- Combination of strict Ref/Def and Initialization
- Only one file has definition
- Definition is declaration without storage class specifier or `extern` with initializer
- Having multiple definitions causes undefined behavior!
  (does not mean that you get an error message!)

Linkage Models

The Standard

- Combination of strict Ref/Def and Initialization
- Only one file has definition
- Definition is declaration without storage class specifier or `extern` with initializer
- Having multiple definitions causes undefined behavior! (does not mean that you get an error message!)

Conclusion

- False assumption on linkage model can be source of bugs!
- gcc under Linux does not use the standard model, but the UNIX one
- Do not rely on that when you want to write portable code!

Linkage Models

The Standard

- Combination of strict Ref/Def and Initialization
- Only one file has definition
- Definition is declaration without storage class specifier or `extern` with initializer
- Having multiple definitions causes undefined behavior! (does not mean that you get an error message!)

Conclusion

- False assumption on linkage model can be source of bugs!
- gcc under Linux does not use the standard model, but the UNIX one
- Do not rely on that when you want to write portable code!

Guideline

- Use strict Ref/Def
- Exactly one file with definition (declaration without storage class)
- All other declarations use `extern`

C++ Compatibility

- Your C code might be used by a C++ project. Be prepared for that.
- Biggest issue: C++ mangles function names

```
extern int my_func(double, const char *);

int main() {
    return my_func(2.345, "Hallo");
}
```

- Mangled symbol name: `__Z7my_funcdPKc` (GCC 4.0.1)
- Just including a C header will apply mangling to C declarations
- Linker will not be able to find symbols
- Surround declarations in `.h` files with

```
#ifdef __cplusplus
extern "C" {
#endif
/* Here go the C declarations */
#ifdef __cplusplus
}
#endif
```

- Then, the C++ compiler knows that those are C declarations

Memory Layout (Unix/Linux)

- When executed, the memory of a C program is composed into several segments
- Text
  - Executable code
  - Might be read-only to forbid accidental self-modification
- Initialized Data
  - global/local linkage data that has been initialized
  - can be set read-only for `const` variables
- Uninitialized Data (BSS)
  - global/local linkage data that has not been initialized
  - Is initialized with 0 by the kernel at load time
  - No space in the binary needs to be wasted

Memory Layout (Unix/Linux)

. . . cont'd

- Heap
  - Dynamically allocated data (`malloc`)
  - Usually grows upwards
- Stack
  - `auto` variables
  - stack frames
  - spilled registers
  - usually grows downwards
- Know where the segments start
- Gives you an idea where your pointers point to
- Example:
  - You debug and some pointer `0xe502f` segfaults
  - This address is strange (below data, heap, and stack)
  - You must have overwritten the pointer's contents

Memory Layout (Unix/Linux)

...cont'd

- Heap
  - Dynamically allocated data (`malloc`)
  - Usually grows upwards
- Stack
  - `auto` variables
  - stack frames
  - spilled registers
  - usually grows downwards

- Know where the segments start
- Gives you an idea where your pointers point to
- Example:
  - You debug and some pointer `0xe502f` segfaults
  - This address is strange (below data, heap, and stack)
  - You must have overwritten the pointer's contents

Homework

Write a small program that prints the addresses of the segments