Understand how to combine C++ and C in the same program

References:

1. C++ Primer, 5th Edition, Stanley Lippman, Josee Lajoie, Barbara Moo.
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup.
3. Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition, Scott Meyers.
4. More Effective C++: 35 New Ways to Improve Your Programs and Designs, Scott Meyers.

- In many ways, the things you have to worry about when making a program out of some components in C++ and some in C are the same as those you have to worry about when cobbling together a C program out of object files produced by more than one C compiler.
  - There is no way to combine such files unless the different compilers agree on implementation-dependent features such as:
    - the size of `int`s and `double`s,
    - the mechanism by which parameters are passed from caller to callee,
    - and whether the caller or the callee orchestrates the passing.
  - These pragmatic aspects of mixed-compiler software development are quite properly ignored by language standardization efforts.
    - So, the only reliable way to know that object files from compiler's A and B can be safely combined in a program is to obtain assurances from the vendors of A and B that their products produce compatible output.
- This is as true for programs made up of C++ and C as it is for all-C++ or all-C programs
  - So before you try to mix C++ and C in the same program, make sure your C++ and C compilers generate compatible object files.
  - Having done that, there are four other things you need to consider: ***name mangling***, ***initialization of statics***, ***dynamic memory allocation***, and ***data structure compatibility***.

Name Mangling

- This is the process through which C++ compilers give each function in the program a unique name.
  - In C, this process is unnecessary, because you can't overload function names, but nearly all C++ programs have at least a few functions with the same name.
  - For example, the `iostream` library declares several versions of `operator<<` and `operator>>`.
  - Overloading is incompatible with most linkers, because linkers generally take a dim view of multiple functions with the same name.
  - Name mangling is a concession to the realities of linkers; in particular, to the fact that linkers usually insist on all function names being unique.
- As long as you stay within the confines of C++, name mangling is not of concern.
  - If you have a function name ***draw_line*** that a compiler mangles into ***xyzzyx_draw_line***, you'll always use the name ***draw_line***, and you'll have little reason to care that the underlying object files happen to refer to ***xyzzyx_draw_line***.
- It is a different story if ***draw_line*** is in a C library.
  - In that case, your C++ source file probably includes a header file that contains this declaration:

    void draw_line(int x0, int y0, int x1, int y1);

    and your code contains calls to ***draw_line*** in the usual fashion. Each such call is translated by your compilers into a call to the mangled name of that function, so that when you write this:

    draw_line(a, b, c, d);

your object files contain a function call that corresponds to this: ***xyzzyx_draw_line(a, b, c, d);***

- But if ***draw_line*** is a C function, the object file (or archive, or dynamically linked library, etc.) that contains the compiled version of ***draw_line*** contains a function called ***draw_line***; no name mangling has taken place.
- When you try to link the object files comprising your program together, you'll get an error, because the linker is looking for a function called ***xyzzyx_draw_line***, and there is no such function.

- To solve this problem, you need a way to tell your C++ compilers not to mangle certain function names.
  - You never want to mangle the names of functions written in other languages, whether they be in C, assembler, FORTRAN, Lisp, Forth, etc.
  - After all, if you call a C function named ***draw_line***, it's really called ***draw_line***, and your object code should contain a reference to that name, not to some mangled version of that name.
- To suppress name mangling, use C++'s **extern "C"** directive:

```
/* declare a function called draw_line; don't mangle its name */
extern "C"
void draw_line(int x0, int y0, int x1, int y1);
```

  - Don't assume that where there's an **extern "C"**, there must be an **extern "Pascal"** and an **extern "FORTRAN"** as well. There's not, at least not in the standard.
  - The best way to view **extern "C"** is not as an assertion that the associated function is written in C, but as a statement that the function should be called as if it *were* written in C.
  - Technically, **extern "C"** means that the function has C linkage.
    - But what that means is far from clear.
    - One thing it always means, however, is that name mangling is suppressed.
  - For example, if you have to write a function in assembler, you could declare it **extern "C"**, too:

```
/* this function is in assembler - don't mangle its name */
extern "C"
void twiddle_bits(unsigned char bits);
```

- You can even declare C++ function **extern "C"**.
  - This can be useful if you're writing a library in C++ that you'd like to provide to clients using other programming languages.
  - By suppressing the name mangling of your C++ function names, your clients could use the natural and intuitive names you choose instead of the mangled names your compilers would otherwise generate:

```
/* the following C++ function is designed for use outside C++
   and should not have its name mangled */
extern "C" void simulate_rope(int iterations);
```

- Often you'll have a slew of functions whose names you don't want mangled, and it would be a pain to precede each with **extern "C"**.
  - Instead, **extern "C"** can also be made to apply to a set of functions by enclosing them all in curly braces:

```
extern "C" { /* disable name mangling for all the following functions */
    void draw_line(int x0, int y0, int x1, int y1);
    void twiddle_bits(unsigned char bits);
    void simulate_rope(int iterations);
    ...
}
```

- This use of **extern "C"** simplifies maintenance of header files that must be used with both C++ and C.
  - When compiling for C++, you'll want to include **extern "C"**, but when compiling for C, you won't.

- By taking advantage of the fact that the preprocessor symbol **__cplusplus** is defined only for C++ compilations, you can structure your polygot header files as follows:

```
#ifdef __cplusplus

extern "C" { /* disable name mangling for all the following functions */

#endif

    void draw_line(int x0, int y0, int x1, int y1);
    void twiddle_bits(unsigned char bits);
    void simulate_rope(int iterations);
    ...

#ifdef __cplusplus
}
#endif
```

- There is no such thing as a "standard" name mangling algorithm - different compilers are free to mangle names in different ways, and different compilers do.

Initialization of Statics

- You need to deal with the fact that in C++, lots of code can get executed before and after ***main***.
  - This is in direct opposition to the way we normally think about C++ and C programs, in which we view ***main*** as the entry point to execution of the program.
  - In particular, ***static initialization*** - the initialization of static class objects and objects at global, namespace, and file scope - occurs before the body of ***main*** is executed.
  - Similarly, objects that are created through static initialization must have their destructors called during ***static destruction***; that process typically takes place after ***main*** has finished executing.
- How to resolve the dilemma that ***main*** is supposed to be invoked first, yet objects need to be constructed before ***main*** is executed?
  - Many compilers insert a call to a special compiler-written function at the beginning of ***main*** which takes care of static initialization.
  - Similarly, compilers often insert a call to another special function at the end of ***main*** to take care of the destruction of static objects.
  - Code generated for ***main*** often looks as if ***main*** had been written like this:

```
int main(int argc, char* argv[])
{
    perform_static_initialization();      /* generated by the C++ implementation */

    the statements you put in main go here;

    perform_static_destruction();         /* generated by the C++ implementation */
}
```

- You should try to write ***main*** in C++ if you write any part of a software system in C++.
  - If a C++ compiler adopts the above approach to the initialization and destruction of static objects, such objects will neither initialized nor destroyed unless ***main*** is written in C++.
  - Sometimes it would seem to make more sense to write ***main*** in C - say if most of a program is in C and C++ is just a support library.

- Nevertheless, there's a good chance the C++ library contains static objects (or it might in the future), so it's still a good idea to write ***main*** in C++ if you possibly can.
- That doesn't mean you need to rewrite your C code, however. Just rename the ***main*** you wrote in C to be ***real_main***, then have the C++ version of ***main*** call ***real_main***:

```
extern "C"
int real_main(int argc, char* argv[]);      /* implement this function in C */

int main(int argc, char* argv[])            /* write this in C++ */
{
    return real_main(argc, argv);
}
```

- If you cannot write ***main*** in C++, you've got a problem, because there is no other portable way to ensure that constructors and destructors for static objects are called.

Dynamic Memory Allocation

- The general rule is simple: the C++ parts of a program use **new** and **delete**, and the C parts of a program use ***malloc*** (and its variants) and ***free***.
- As long as memory that came from **new** is deallocated via **delete** and memory that came from ***malloc*** is deallocated via ***free***, all is well.
- Calling ***free*** on a **new**ed pointer yields undefined behavior, however, as does **delete**ing a ***malloc***ed pointer.
- The only thing to remember, then, is to segregate rigorously your **new**s and **delete**s from your ***malloc***s and ***free***s.
- Sometimes this is easier said than done. Consider the handy ***strdup*** function, which, though standard in neither C nor C++ is nevertheless widely available:

```
char* strdup(const char* ps);       /* return a copy of the string pointed to by ps */
```

  - If a memory leak is to be avoided, the memory allocated inside ***strdup*** must be deallocated by ***strdup***'s caller.
  - But how is the memory to be deallocated? By using **delete**? By calling ***free***?
  - If the ***strdup*** you're calling is from a C library, it's the latter. If it was written for a C++ library, it's probably the former.
  - What you need to do after calling ***strdup***, then, varies not only from system to system, but also from compiler to compiler.
  - To reduce such portability headaches, try to avoid calling functions that are neither in the standard library nor available in a stable form on most computing platforms.

Data Structure Compatibility

- Can data be passed between C++ and C programs?
- There's no hope of making C functions understand C++ features, so the level of discourse between the two languages must be limited to those concepts that C can express.
  - It should be clear there's no portable way to pass objects or to pass pointers to member functions to routines written in C.
  - C does understand normal pointers, however, so, provided your C++ and C compilers produce compatible output, functions in the two languages can safely exchange pointers to objects and pointers to non-member functions or static functions.

    - Naturally, structs and variables of built-in types (e.g., **int**s, **char**s, etc.) can also freely cross the C++/C border.
- Because the rules governing the layout of a **struct** in C++ are consistent with those of C, it is safe to assume that a structure definition that compiles in both languages is laid out the same way by both compilers.
    - Such structs can be safely passed back and forth between C++ and C.
    - If you add *nonvirtual* functions to the C++ version of the struct, its memory layout should not change, so objects of a struct (or class) containing only non-virtual functions should be compatible with their C brethren whose structure definition lacks only the member function declarations.
    - Adding *virtual* functions ends the game, because the addition of virtual functions to a class causes objects of that type to use a different memory layout.
    - Having a struct inherit from another struct (or class) usually changes its layout, too, so structs with base structs (or classes) are also poor candidates for exchange with C functions.

Summary

- If you want to mix C++ and C in the same program, remember the following simple guidelines:
    - Make sure the C++ and C compilers produce compatible object files.
    - Declare functions to be used by both languages **extern "C"**.
    - If at all possible, write ***main*** in C++.
    - Always use **delete** with memory from **new**; always use ***free*** with memory from ***malloc***.
    - Limit what you pass between the two languages to data structures that compile under C; the C++ version of structs may contain non-virtual member functions.