(C++)--

Type System

- C has no bool type. Instead, we normally use int to play the role of bool. Remember, zero is interpreted as false and anything else is true. We can "inform" C of a bool type by using the following macros.

  ```
  #define bool int
  #define true 1
  #define false 0
  ```

- C has no reference parameters. We'll eventually use pointers instead; that's how reference parameters are implemented anyway. It is also possible to use macros to simulate passing by reference for short functions.

- C has no classes and no member functions. Although C does offer struct, it is just used to group a collection of related data items together. It can't contain member functions and it can't really encapsulate its data. In places where we used to use member functions, we will just have to use regular functions (sometimes called free functions) instead.

  Note: Defining a new struct in C does no automatically let you use the name of the struct by itself as a new type name. For example, if you have a struct named Point, you have to say struct Point to talk about its type. If you like, you can use typedef to give a short name to a user-defined type. Also notice that structs can't be passed by value.

- C has no virtual functions. Ordinarily we use virtual functions to do things like let one part of our program call a function without knowing what function, in particular, it's calling. On occasions when we need this kind of run-time polymorphism, we can use pointers to functions instead. We can store a pointer to a function in a variable or a field and then call the function when we need to. Remember that the name of a function evaluates to the address where it's stored. The most difficult part is writing out the type of a pointer to a function.

Definitions and Declarations

- The const keyword in C behaves slightly differently. You can't use it to define values that are treated as compile-time constants. This matters because, in many places, array size is expected to be a compile-time constant. The C compiler won't accept code like the following:

  ```
  const int SIZE = 200;
  int plist [ SIZE ];
  ```

  For constants that must be treaded as compile-time constants, we have to use #define instead, like:

  ```
  #define SIZE 200

  int plist [ SIZE ];
  ```

- C has no inline function support. However, in a pinch one can use #define to do the same job.

- C has no user-defined operators. We just have to write functions to do the same job.

- C does not permit function overloading. We just have to make sure that different functions have different names.

Miscellaneous

- C has no general notion of value semantics. This means that we can't expect to assign structs by value or pass and return them from functions by value. This is almost always done by passing addresses in Cand has the advantage of generally being a lot faster anyway. Using const can help you to keep up with whether or not a function is permitted to modify the value pointed to by one of its parameters. If you really need to make a deep copy of a user-defined type, you can write your own function to do it. Alternatively, if the type doesn't contain any pointers, you can probably get memcpy() to do the work for you.

- Some C compilers do not support // - style comments from C++. This is only an inconvenience since we can get by with /* …. */ instead.

Standard Library

- C has no special-purpose string type. We'll have to use null-terminated character strings instead.

- C does not support C++ I/O streams. A C programmer will typically use the C standard I/O library instead.

- C has not Standard Template Library (STL). This is particularly inconvenient when we need a data structure to accommodate an arbitrary number of items. In these cases, we have to just manage the resizing of the data structures ourselves. Popular choices are to use linked lists or dynamically sized arrays allocated an reallocated as needed.