1. Compile cleanly at high warning levels.

Summary

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

Discussion

Your compiler is your friend. If it issues a warning for a certain construct, often there's a potential problem in your code.

Successful builds should be silent (warning-free). If they aren't, you'll quickly get into the habit of skimming the output, and you *will* miss real problems. (See Item 2.)

To get rid of a warning: a) understand it; and then b) rephrase your code to eliminate the warning and make it clearer to both humans and compilers that the code does what you intended.

Do this even when the program seemed to run correctly in the first place. Do this even when you are positive that the warning is benign. Even benign warnings can obscure later warnings pointing to real dangers.

Examples

Example 1: A third-party header file. A library header file that you cannot change could contain a construct that causes (probably benign) warnings. Then wrap the file with your own version that **#include**s the original header and selectively turns off the noisy warnings for that scope only, and then **#include** your wrapper throughout the rest of your project. Example (note that the warning control syntax will vary from compiler to compiler):

```
// File: myproj/my_lambda.h -- wraps Boost's lambda.hpp
//   Always include this file; don't use lambda.hpp directly.
//   NOTE: Our build now automatically checks "grep lambda.hpp <srcfile>".
// Boost.Lambda produces noisy compiler warnings that we know are innocuous.
// When they fix it we'll remove the pragmas below, but this header will still exist.
//
#pragma warning(push)       // disable for this header only
 #pragma warning(disable:4512)
 #pragma warning(disable:4180)
 #include <boost/lambda/lambda.hpp>
#pragma warning(pop)        // restore original warning level
```

Example 2: "Unused function parameter." Check to make sure you really didn't mean to use the function parameter (e.g., it might be a placeholder for future expansion, or a required part of a standardized signature that your code has no use for). If it's not needed, simply delete the name of a function parameter:

```
// ... inside a user-defined allocator that has no use for the hint ...

// warning: "unused parameter 'localityHint'"
pointer allocate( size_type numObjects, const void *localityHint = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) ) );
}

// new version: eliminates warning
pointer allocate( size_type numObjects, const void * /* localityHint */ = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) )   );
}
```

Example 3: "Variable defined but never used." Check to make sure you really didn't mean to reference the variable. (An RAII stack-based object often causes this warning spuriously; see Item 13.) If it's not needed, often you can silence the compiler by inserting an evaluation of the variable itself as an expression (this evaluation won't impact run-time speed):

```
// warning: "variable 'lock' is defined but never used"
void Fun() {
  Lock lock;

  // ...

}

// new version: probably eliminates warning
void Fun() {
  Lock lock;
  lock;

  // ...

}
```

Example 4: "Variable may be used without being initialized." Initialize the variable (see Item 19).

*Example 5: "Missing **return**."* Sometimes the compiler asks for a **return** statement even though your control flow can never reach the end of the function (e.g., infinite loop, **throw** statements, other **return**s). This can be a good thing, because sometimes you only *think* that control can't run off the end. For example, **switch** statements that

do not have a **default** are not resilient to change and should have a **default** case that does **assert(false)** (see also Items 68 and 90):

```
// warning: missing "return"
int Fun( Color c ) {
  switch( c ) {
  case Red:    return 2;
  case Green:  return 0;
  case Blue:
  case Black:  return 1;
  }
}

// new version: eliminates warning
int Fun( Color c ) {
  switch( c ) {
  case Red:    return 2;
  case Green:  return 0;
  case Blue:
  case Black:  return 1;
  default:     assert( !"should never get here!" );   // !"string" evaluates to false
               return -1;
  }
}
```

Example 6: "Signed/unsigned mismatch." It is usually not necessary to compare or assign integers with different signedness. Change the types of the variables being compared so that the types agree. In the worst case, insert an explicit cast. (The compiler inserts that cast for you anyway, and warns you about doing it, so you're better off putting it out in the open.)

Exceptions

Sometimes, a compiler may emit a tedious or even spurious warning (i.e., one that is mere noise) but offer no way to turn it off, and it might be infeasible or unproductive busywork to rephrase the code to silence the warning. In these rare cases, as a team decision, avoid tediously working around a warning that is merely tedious: Disable that specific warning only, disable it as locally as possible, and write a clear comment documenting why it was necessary.

References

[Meyers97] §48 • [Stroustrup94] §2.6.2