C++ Pitfalls

wheeler@kde.org

Home
TagLib
JuK
C++ Pitfalls
Wallpapers
Presentations
Bugs & Feature Requests
Brief Bio
Books

Sections

References
Const-ness
Public, Protected and Private
Virtual Member Functions
Mutable Variables

Links

Binary Compatibility with C++
Parameter Passing Rules
KDE Programming Tips

This is an attempt to provide an overview of many of the C++ pitfalls that beginner to moderately experienced C++ programmers often fail to understand. Specifically this addresses mistakes that I've seen from newer KDE contributers over the last couple of years. Please note that this is not an attempt to replace a good C++ reference, but simply to provide an introduction to some often misunderstood concepts and to point out their usefulness.

References

References are a way of assigning a "handle" to a variable. There are two places where this is used in C++. We'll discuss both of them briefly.

Assigning References

This is the less often used variety of references, but still worth noting as an introduction to the use of references in function arguments. Here we create a reference that looks and acts like a standard C++ variable except that it operates on the same data as the variable that it references.

```
int foo = 3;     // foo == 3
int &bar = foo;  // foo == 3
bar = 5;         // foo == 5
```

Here because we've made *bar* a reference to *foo* changing the value of bar also changes the value of *foo*.

Passing Function Arguments With References

The same concept of references is used when passing variables. For example:

```
void foo( int &i )
{
    i++;
}

int main()
{
    int bar = 5;   // bar == 5
    foo( bar );    // bar == 6
    foo( bar );    // bar == 7

    return 0;
}
```

Here we display one of the two common uses of references in function arguments -- they allow us to use the conventional syntax of passing an argument by value but manipulate the value in the caller.

Note: While sometimes useful, using this style of references can sometimes lead to counter-intuitive code. It is not clear to the caller of *foo()* above that *bar* will be modified without consulting an API reference.

However there is a more common use of references in function arguments -- they can also be used to pass a handle to a large data structure without making multiple copies of it in the process. Consider the following:

```
void foo( const std::string &s )
{
    std::cout << s << std::endl;
}

void bar( std::string s )
{
    std::cout << s << std::endl;
}

int main()
{
    std::string text = "This is a test.";

    foo( text ); // doesn't make a copy of "text"
    bar( text ); // makes a copy of "text"

    return 0;
}
```

In this simple example we're able to see the differences in *pass by value* and *pass by reference*. In this case pass by value just