Doc No: N1741=04-0181

Project: Programming Language C++

Date: Friday, November 05, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

Proposal for Extending the switch statement

Note this paper is a preliminary discussion document with the intent that we should determine how far we wish to go in providing an extended `switch`-statement for C++.

Background

The `switch`-statement in C++ has been inherited unaltered from C. In C restricting the selection to cases that were constant integral values made sense because the integer types are the only ones that can be sensibly and consistently compared for equality. Floating point types can be compared but there are well known reasons for comparison for equality to be unreliable. Comparing pointers in the context of a selection statement would almost always not be what the programmer intended. For example, if allowed:

```
char * array;
/* code that initializes array */
switch(array){
     case "one": /* action */
                 break;
     case "two": /* action */
                 break;
     case "three": /* action */
                 break;
     case "four": /* action */
                 break;
     default: /* action */
}
```

would normally be intended to compare the strings not the addresses. C does not support the equality comparison for user-defined types (other than `enum` via decay to integer values).

Again the restriction to `const` integer expressions made a good deal of sense in the C context because it allowed other mechanisms for selection other than a straight linear search.

For the programmer, the `switch`-statement has two distinct things to offer. The first is that it provides syntactic sugar for a series of `if…else if` statements. We should not ignore the value of that, a `switch`-statement can make the logic of a program clearer and easier to maintain.

The second feature of the current `switch`-statement is that it enables the compiler to optimize the selection by a variety of different techniques. The current requirements for `switch`-statements ensures that such optimizations are possible at compile time.

The `switch`-statement we have today is a valuable programming tool. However many programmers are frustrated by the constraints that have been inherited from C. In C++ exact comparisons can be, and often are, provided for user defined types. That means that the continued restriction that the control expression must be convertible to `int` makes much less sense. For example:

```
std::string select_on;
// code establishing a value for select_on
switch(select_on){
     case "one": /* action */
               break;
     case "two": /* action */
               break;
     case "three": /* action */
               break;
     case "four": /* action */
               break;
     default: /* action */
}
```

Is expressive, unambiguous and easy to maintain. The single problem is that it is also not allowed. Instead the programmer has to write:

```
std::string select_on;
// code establishing a value for select_on
     if(select_on == "one") {
          /*action */
     }
     else if(select_on == "two"){
          /*action */
     }
     else if(select_on == "three"){
          /*action */
     }
     else if(select_on == "four"){
          /*action */
     }
     else /* action */
```

If the actions are anything but simple statements the code can become a maintenance nightmare.

Even from the programmer's perspective reformulating the constraints on the `switch`-statement to allow its use as an alternative to nested `if … else` statements makes a great deal of sense.

The question is to what degree should we loosen the constraints. It seems to me that there are three possible degrees of loosening all based on using the `operator ==` for the type of the control expression (others want to consider other comparisons or the allowance of case labels based on ranges).

1) Allow the cases to be `const` pure expressions. That is expressions that are free from side-effects (so re-ordering of their evaluation has no implications to the program as a whole). Being `const` expressions would allow a compiler to evaluate them should the implementer support compile time evaluation for the relevant expressions. The compiler could then set up optimized selection. The above example is a case in point. High quality implementations could select efficiently on the basis of string literals and a `std::string` control expression.

2) Allow pure expressions that are not `const` and so cannot be evaluated statically. Because the expressions are pure, the compiler still has some potential for re-ordering the evaluations but generally this version of an extended `switch`-statement would be implemented by transformation into the equivalent set of nested `if…else` statements. This version would be useful, for example, for an internationalized program where the cases would be read from a suitable data file at execution time in the chosen language.

3) Allow any expressions convertible to the type of the control expression. In this case the `switch`-statement would become pure syntactic sugar for a series of nested `if…else` statements.

Proposal

Expand the grammar of C++ to include:

pure-expression

const-pure-expression

Expand the specification of a `switch`-statement or provide a new alternative 'selection statement' so:

The control expression can have any type for which there is an `operator==` provided.

Specify that any necessary conversions will be limited to those that convert the type of the case label expression to that of the control expression.

The case labels shall be `const`-pure-expressions of the same type as the control expression or convertible to the type of the case label expression.

Consider a further extension to include case labels that are pure-expressions and specify that where two or more case labels compare equal to the control expression that the one selected is unspecified.

Discussion

The definition of a pure-expression at first sight seems easy to provide by extending the grammar of the language. Unfortunately operators that are essentially pure for fundamental types are not required to not retain that quality when overloaded for udts.

For example `operator <<` is 'pure' for fundamental types but is not when overloaded for `std::ostream`.

This means that we need to provide a semantic rather than a syntax (grammar) specification of what it means to be a pure expression.

A `const`-pure-expression is a pure-expression whose evaluation only accesses `const` objects. We will need to consider whether we wish to limit this to compile time `const` expressions as opposed to execution time `const` expressions. For example, the difference between shows the two concepts (a compile time `const` expression only uses values known at compile time):

```
int const value(0);  // compile time const
```

and

```
int foo();
int const value(foo());  // execution time const.
```

I think that even the most limited extension to allow the control expression of a `switch`-statement to be any type supporting `operator ==` with case labels being compile time `const`-pure-expressions would be helpful to many programmers as well as allowing better maintenance and better optimization of source code. The 'pure' requirement allows the compiler more opportunities for optimization but until we have a mechanism for declaring functions as pure it greatly limits the expressions that can be used in case labels unless we are willing to allow it to be unspecified if and when the side effects of evaluating such expressions will occur.

Allowing execution time `const`-pure-expressions in case labels or even pure-expressions in case labels would further expand the utility to many programmers but with reduced optimization opportunities.

It would probably be advantageous to warn programmers if the type of the expression in a case label reduces potential optimization but that might be considered a QoI issue.

A case can be made for two other modifications to the C++ `switch`-statement:

Allow case labels to specify ranges. This would be useful for selecting on types that are completely ordered – particularly useful for floating point types.

Deprecate the use of 'fall through' to the first `break`-statement and provide an alternate syntax where 'fall through' does not happen.

Doing the second without the providing a mechanism for some form of range case label would add problems to some coding cases. Fall-through is currently used as a substitute for range or set case labels.

An option that could be considered is that of adding syntax to allow a case label to reference a container for objects of the same type as the control expression. Something like:

```
int selector;// i.e. get a selection value
std::vector<int> values1;
std::vector<int> values2;
//code initializing the above variables
```

```
switch (selector){
     case 1: // actions
          break;
     case values1: // actions
          break;
     case values2: // actions
          break;
     default: //actions
}
```

If we can come up with some suitable qualifier to distinguish the new switch from the classical one then we could also remove the fall through so the above code could be simplified to:

```
int selector;// i.e. get a selection value
std::set<int> values1;
std::set<int> values2;
//code initializing the above variables
new switch (selector){
     case 1: // actions
     case values1: // actions
     case values2: // actions
     default: //actions
}
```

However this mechanism will not address the problem of floating point types where comparison for equality is inexact.

It is worth noting that ranges can be supported by udts. For example:

```
struct float_range{
     float upper;
     float lower;
     float_range(float l, float u): upper(u), lower(l){}
     float_range(float l):upper(l), upper(u){}
}
bool operator==(float f, float_range const& fr){
     return (f<fr.upper && f>fr.lower)? true : false;
}
```

However the cost of supporting ranges that way is that we have an asymmetric equality operator. We would also have to revisit the requirements for the operands of the `operator==` used in the `switch`-statement to allow for this asymmetry. The allowance of a case label using a container such as `std::set` would work well for small finite ranges but supporting asymmetric equality comparisons provides for a greater range of uses.

Another option would be to provide a new selection syntax in which both a selector value and a comparison operator was provided.

A set of use cases might help us choose how far we want to extend the `switch`-statement.