ANSI Doc No: X3J16/93-0143
ISO Doc No: WG21/N0350
Date: September 27, 1993
Project: Prog. Language C++
Reply To: Dag Brück
dag@dynasim.se

Extension proposal: Boolean data type

Dag Brück[1] and Andrew Koenig[2]

1. Introduction

The idea of a Boolean datatype in C++ is a religious issue. Some people, particularly those coming from Pascal or Algol, consider it absurd that C should lack such a type, let alone C++. Others, particularly those coming from C, consider it absurd that anyone would bother to add such a type to C++.

Why bother indeed? The main reason is that people who like such types often write header files that contain things like this:

```
typedef int bool;

class Mine {
    // ...
};

bool operator==(const Mine &, const Mine &);
```

Such a header file is just fine until someone else's header file says

```
typedef char bool;
```

If both header files used `int`, there would be no problem, but trying to define `bool` as two different types just won't work.

Several widely used C++ libraries, including X11 and InterViews, define a Boolean datatype. The differences in the definitions of the Boolean datatype create very real compatibility problems when mixing libraries from multiple sources.

In principle, this problem could be solved by agreeing on a single type definition of `bool` as part of the standard library, just as `NULL` is part of

1 Author's address: Dynasim AB, Research Park Ideon, S-223 70 Lund, SWEDEN. E-mail: dag@dynasim.se

2 Author's address: AT&T Bell Laboratories, 600 Mountain Avenue, Room 6D-416B, PO Box 636, Murray Hill, NJ 07974-0636, USA. E-mail: ark@research.att.com

the standard C library. However, that approach has several significant disadvantages when compared with the proposal that follows. In particular, any proposal that makes the Boolean type a synonym for any other built-in type gives up the possibility of overloading on that type.

On the surface, it is hard to believe that any compromise is possible between those extremes. After all, a language either has a Boolean type or it does not. However, closer examination suggests that a middle ground might be possible: a Boolean type whose existence the programmer has the option to ignore.

This proposal shows how such a type can work. It is the result of discussions with a number of people, both in person and via electronic mail. Of course, any mistakes or misjudgments in this proposal are ours alone.

2. Executive summary

We propose the creation of a new built-in type called `bool`, with the following properties:

- `bool` is a unique signed integral type, just as the `wchar_t` is a unique unsigned type.
- There are two built-in constants of this type: `true` and `false`.
- Operator `++` is defined for `bool`; it always sets its operand to `true`. This operator is an anachronism, and its use is deprecated.
- A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero.
- A numeric or pointer value is automatically converted to `bool` when needed. This is an anachronism, and its use is deprecated.
- When converting a numeric or pointer value to `bool`, a zero value becomes `false`; a non-zero value becomes `true`.
- The built-in operators `&&`, `||`, and `!` are changed to take `bool` values as their arguments and return `bool` results.
- The relational operators `<`, `>`, `<=`, `>=`, `==`, and `!=` are changed to yield a `bool` result.
- Expressions used as conditions in `if`, `for`, `while`, or `do` statements or as the first operand of a `?:` expression, are automatically converted to `bool`.

We believe this extension is upward compatible except for the new keywords `bool`, `true`, and `false`.

3. Rationale and examples

Current practice

A quick investigation of one computer system reveals that Boolean datatypes are quite common in widely used libraries. In the `/usr/include` area on one of our machines (`bellman.control.lth.se`), the definitions of Boolean in Table 1 can be found. Various libraries also define `true` and `false`, see Table 2. The header files that do not rely on a definition of Boolean seem to be part of the Sun operating system.

curses.h	#define bool char
X11	#define Bool int
gcc/X11	Multiple definitions of Boolean, depends on machine type etc.
netap/appletalk.h	typedef int boolean;
rpc/types.h	#define bool_t int
rpc/*.h	Many uses of bool_t
nfs/nfs.h	Uses bool_t
tfs/tfs.h	Uses bool_t
sunwindow/rect.h	#define bool unsigned
sunwindow/sun.h	typedef enum {False = 0, True = 1} Bool;
sunwindow/*.h	Uses bool
suntool/*.h	Uses bool, at least as comment
InterViews	typedef unsigned boolean;
NEWMAT matrices	`bool` defined as class or char, depends on compiler
C++Views	`boolean` defined with typedef
sys/types.h (Solaris)	typedef'd as enumeration

Table 1. Current definitions of `boolean`.

curses.h	#define TRUE 1 #define FALSE 0
X11/Intrinsic.h	(same)
gcc/X11/Intrinsic.h	(same)
netat/appletalk.h	(same)
pixrect/gtvar.h	(same)
sbusdev/audiovar.h	(same)
suntool/*.h (2 places)	(same)
sunwindow/ (2 places)	(same)
scsi/adapters/espvar.h	(same)
X11 (2 places)	#define True 1 #define False 0
rpc/types.h	#define FALSE (0) #define TRUE (1)
sundev/dfreg.h	#define TRUE 1 (but not FALSE!)
sunwindow/sun.h	typedef enum {False = 0, True = 1} Bool;
InterViews	static const unsigned true = 1; static const unsigned false = 0;
Rogue Wave Matrix	(same as curses.h)
C++Views	(same as curses.h)
sys/types.h (Solaris)	B_FALSE and B_TRUE enumerators

Table 2. Current definitions of `true` and `false`.

We have also found the following definition of Boolean in sys/types.h under Solaris 2.2:

```
typedef enum boolean {B_FALSE, B_TRUE} boolean_t;
```

with a reference to POSIX. It is however unclear if this definition is part of POSIX or a Sun extension.

Our conclusion is that the boolean datatype is a fact of life whether it is part of the C++ standard or not. However, there is currently a substantial variation in the definitions, as shown by the tables. These variations create several compatibility problems:

- Different names (e.g., `bool`, `Bool`, `boolean`, `bool_t`) are used to identify one conceptual type.
- The Boolean datatype may be redefined, which in the case of typedefs is a fatal compilation error.
- Different definitions may not be type compatible, for example, Boolean defined as an `int` and as an enumeration. Overloading on Boolean types is particularly sensitive.
- Different amounts of storage may be allocated for Boolean types.

C has a less strict type system than C++, so these variations are less of a problem in C than in C++. Because C++ does not allow implicit conversion from `int` to enumeration, current definitions of Boolean that use an enumeration cannot be used without a large number of explicit typecasts. In the case of Solaris 2.2, this is particularly worrying because a definition in sys/types.h is likely to frequently used.

Built-in vs. user-defined type

Adding a new type is essential for overloading. For example:

```
void f(int);
void f(bool);

main() { f(3 < 4); }
```

In order for this example to work, `bool` *must* be a built-in or enumeration type; there is simply no way around it.

On the other hand, allowing the standard promotion from `bool` to `int` is essential for backward compatibility:

```
void f(int);
void f(double);

main() { f(3 < 4); }
```

would change meaning otherwise. The point of the standard promotion is to permit any `bool` expression to act precisely like an `int` expression in the absence of explicitly declared `bool` objects.

The Boolean datatype cannot be defined as an ordinary class type because of the rule that says that at most one user-defined conversion is automatically applied:

```
class X {
public:
    X(int);
};

void f(X);

main()
{
    f(3 < 4);
}
```

For `f(3<4)` to work, the conversion from `bool` to `int` must not be a user-defined conversion.

Conversion from int to bool

For similar reasons, conversion from numeric or pointer types to `bool` must be allowed. We do not expect people to stop using `int` values to hold flags, which means that things like

```
int done = 0;

while (!done) {
    // ...
    if (/* ... */)
        ++done;
    // ...
}
```

must remain legal. In this example, the expression `!done` causes `done` to be converted to `bool` before negation, with exactly the same semantics as always.

Another common idiom, which we probably must keep in the name of programmer compatibility, is the following:

```
int* p;
if (p)
    *p = 42;
```

While some people might wish to receive warnings about this, many others would not.

Why should bool be a *signed* integral type when the only values are false (zero) and true (one)? The answer is that otherwise it would have to be promoted to `unsigned int`, which would probably break code:

```
void f(int);
void f(unsigned int);

f(3 < 4); // must call f(int)
```

Boolean as magic enumeration

We also considered defining Boolean as a magic built-in enumeration defined in global scope. This approach is quite attractive at first sight, for example, it has the advantage that the new reserved words can be re-defined in a local scope (we are not certain this really is an advantage, though).

We have abandoned this approach, mainly because of the unclear relationship to the underlying type of this magic enumeration. As an ordinary enumeration the underlying type would be unsigned, but as explained above, the promoted type really must be `int`.

Operations defined on bool

Expressions like `b1+b2` must produce an int because it would otherwise cause a C incompatibility in

```
int n = (x < y) + (y < z);
// n is the number of true conditions
```

Operators `&`, `|` and `^` could yield a bool result if and only if both operands are bool, but `~` is a problem: in C, `~(3>4)` is -1 so we think we have to keep it that way. That suggests that maybe it's easier to leave `&`, `|` and `^` as pure bitwise operators on integers.

Realistically, people who say `++` today on a Boolean value treat `++a` as `(a=true)`. The real trouble is `--a`, which won't do the same thing as `(a=false)`. Our vote: `++a` means `(a=true)` but is an anachronism; `--a` is ill-formed.

Other concerns

Another issue we have to think about is functions in the standard C library that conceptually return a Boolean value. The problem is bigger if any standard function takes a pointer to an `int` which should be `bool`. We cannot think of things that behave like pointer to Boolean; however, things like `isupper()` have to start returning `bool`.

There is no doubt that the keyword `bool` will step on many programs that say things like

```
typedef int bool;
```

We do not consider this a major drawback. Programs that are actually using `bool` to represent a Boolean type will almost surely continue to work after the offending `typedef` declaration is simply removed. Programs that use `bool` to represent some other type deserve to be taken out and shot.

Here is another, slightly contrived, example of code that will break under this proposal.

```
template <class T>
T& example(T) { static T x; return x; }

main() { int& x = example(3 < 4); }
```

We believe that code of this kind is rare, and a weak argument against the proposal.

We prefer `bool` to `boolean` because C++ generally prefers short type names (int) to longer ones (integer). We prefer `boolean` to `bool` because that is what we really call it, and that is used in many other languages: Ada, Pascal, Modula-2. We prefer either of these to `Bool` or `Boolean` because all other built-in type names are entirely in lower case.

4. Representation

This proposal implicitly makes it difficult to pack Boolean arrays by storing each value in a separate bit because it allows each element of such an array to have its own address. This follows from the usual equivalence between `a+i` and `&a[i]`. The proposal could have been formulated differently, but doing so would surely break programs that define their own `bool` type and use arrays of objects of that type.

Note however that bitfields of type `bool` are allowed because `bool` is an integral type.

5. Conclusions

This proposal is simple and conservative: so much so that some will wonder why it's worthwhile. There are two main reasons: allowing overloading on Boolean expressions and forestalling clashes between libraries that define their own Boolean types.