Why I still use macros in C++

Richard Shepherd

- Basic on Dragon32 as a teenager
- Fortran and Pascal at university
- Modula-2 and Actor at internships
- Fortran and C at Logica
- C++ and Java at Investment banks
- C++ at ESRI R&D Cardiff

Understand your tools and take the time to make things simple

C/C++ preprocesser

Macro definition and application are one of the tasks carried out by the C/C++ preprocessor. Other, unavoidable, preprocessor tasks include: `#include`, `#ifdef` etc, `#pragma` e.g.

```
//talk_macrosInCpp.cpp
#include <iostream>
#include <assert.h>
int main()
{
  int x = 0;
  std::cin >> x;
  assert(x > 0);
  std::cout << x << "\n";
}
```

```
//assert.h
#ifndef __ASSERT_H__
#define __ASSERT_H__
#ifdef NDEBUG
    #define assert(expression) ((void)0)
#else
    #define assert(expression) (void)(                                                \
            (!!(expression)) ||                                                       \
            (_wassert(_CRT_WIDE(#expression), _CRT_WIDE(__FILE__), (unsigned)(__LINE__)), 0) \
        )
#endif //NDEBUG
#endif //__ASSERT_H__
```

Macros

Macros define a text substitution that occurs before the compiler sees the code.

Macros can be evil, despite their good intentions, and should be treated with caution

```
#define PI 3.14159265359   /*Doesn't need initialization of externed shared value*/
#define MAX_NUM_WINDOWS 10
#define SQR(X) X * X              /*Works for multiple types*/
#define MAX(a,b) ((a>b)?(a):(b)) /*Avoid cost of function call*/
```

- PI could be defined multiple times. So you need to undef first.
- No namespacing or scoping, so need to know that MAX_NUM_WINDOWS refers to houses not GUI.
- Safety advice of including brackets

Macro functions are more error prone..

```
#define SQR(X) X * X              /*Works for multiple types*/
int a = 5;
int aSqr = SQR(a); //= 25; //okay
int a1Sqr = SQR(a+1); //= a+1*a+1 = a+a+1 = 11; //not okay!

#define MAX(a,b) ((a>b)?(a):(b)) /*Avoid cost of function call*/
const int limit = 10;
int maxLimitAndA = a;
int numIterations = 0;
for (int i=0; i != limit;)
{
  maxLimitAndA = MAX(maxLimitAndA,i++);
  ++numIterations;
}
std::cout << maxLimitAndA << "\t" << numIterations;
```

They are easily confused with functions, but don't give us the same behavior.

The error messages and debugging are no fun.

Uses are disappearing

```
constexpr double pi = 3.14159265359;

namespace MyGUI
{
  constexpr size_t max_num_windows = 255;
}
class House
{
  static constexpr int max_num_windows = 10;
};

template <typename TVal>
constexpr TVal Sqr(TVal x) { return x * x;}

template <typename TVal1, typename TVal2>
constexpr auto Max(TVal1 x, TVal2 y) { return (x > y) ? x : y;}
```

What's left?

There are a number of useful constants defined, which the preprocessor changes the value of as it proceeds, leaving the current value in the source code, useful for logging and debugging code e.g.:

```
__DATE__ /*Date in "Mmm dd yyyy" format*/
__TIME__ /*Time (starting this compilation unit) in "hh:mm:ss" format*/
__FILE__ /*Fullpath of current source file*/
__LINE__ /*Line number within the current source file*/
__func__ /*Name of current function (not macro)*/
NDEBUG   /*Defined by compiler if not in 'Debug' mode. Relied on by assert*/
```

and..

Power tools: Stringify

```
#define LOG(A) std::cout << #A << ":" << (A) << "\n"
...
  std::string name{"Richard"};
  int x = 10;
  std::cout << "name" << ":" << name << "\n" << "x" << ":" << x;
  LOG(name);
  LOG(x);
```

two ways to output

```
name:Richard
x:10
```

and Concatenate (make new symbolic name)

```
#define CONCAT(A,B,C) A ## _ ## B ## C
...
  float x = 10.0f;
  double y = 25.0;
  auto CONCAT(my,x,y) = x * y;
  std::cout << my_xy;
```

Case study: Making legacy code safer and simpler

Code inherited from another era, that relied on global variables

```
//WARNING destructive routine that alters the shape and messes with global variables, in order to calculate its eroded center
FPoint CalcErodedCenter(Environment& env, const Shape& shape)
{
  const int s_ixpxl1 = env.g_ixpxl1; //Save current values
  const int s_iypxl2 = env.g_ixpxl2;
  const float s_fradb = env.g_fradb;
  ..
  env.g_ixpxl1 = .. //give new values for use by functions that get called
  env.g_fradb = ..
  ..
  if (..)
  {
      ..
      env.g_ixpxl1 = s_ixpxl1; //restore for early return
      env.g_ixpxl1 = s_ixpxl1;
      env.g_fradb = s_fradb;
      return FPoint(0,0);
  }
  ..
  env.g_ixpxl1 = s_ixpxl1; //restore
  env.g_iypxl1 = s_iypxl1;
  env.g_fradb = s_fradb;
  return pnt;
}
```

Assuming we're stuck with the global variables for now, how can we make it safer?

RAII

```
template <typename T>
class Keep
{
  T& ref;
  T val;
public:
  Keep(T& t): ref(t), val(t) {}
  ~Keep() {ref = val;}
};

template <typename T> Keep<T> make_keep(const T& t) { return Keep<T>(t); }

FPoint CalcErodedCenter(Environment& env, const Shape& shape)
{
  const Keep<int> keep_ixpxl1(env.g_ixpxl1); //Automatically restore on exit
  //or const auto& keep_ixpxl1 = make_keep(g_ixpxl1); //Automatically restore on exit
  ..
  env.g_ixpxl1 = ..
  if (..)
  {
      return FPoint(0,0);
  }
  return pnt;
}
```

Restoration is guaranteed and in a known (reverse) order

The KEEP macro

```
#define CONCAT_(a,b) a ## b
#define CONCAT(a,b) CONCAT_(a,b)
#define KEEP(a) const auto& CONCAT(keep,__LINE__) = make_keep(a); CONCAT(keep,__LINE__);

FPoint CalcErodedCenter(Environment& env, const Shape& shape)
{
  KEEP(env.g_ixpxl1); //Automatically restore on exit
  g_ixpxl1 = ..
  if (..)
  {
      return FPoint(0,0);
  }
  return pnt;
}
```

The macro encapsulates the boilerplate and hides knowledge of the name.

A similar example was a macro `FOR_RANGE(element, collection)` until c++11.

(Both were more complex in the days before `auto`)

Before initializer lists

Unit tests benefit from being short, expressive and self describing.

Before c++ initializer-lists this wasn't so easy e.g.

```
double PerimeterLength(const vector<Point>& pnts);

vector<Point> pnts
pnts.push_back(Point(5,10));
pnts.push_back(Point(5,20));
pnts.push_back(Point(15,20));
pnts.push_back(Point(15,5));
EXPECT_EQ(35.0, PerimeterLength(pnts));
```

with less immediately meaningful error messages such as

```
EXPECT_EQ failed: PerimeterLength(pnts) (34.0) != 35.0
```

With initializer lists

this could be:

```
double PerimeterLength(const vector<Point>& pnts);
...
EXPECT_EQ(35.0, PerimeterLength({{5,10},{5,20},{15,20},{15,5}}));
```

where the tests are easy to write with dynamically created data, and can produce easy to read and recognize messages:

```
EXPECT_EQ failed: PerimeterLength({{5,10},{5,20},{15,20},{15,5}}) (34.0) != 35.0
```

Homemade Initializer Lists

```
class MakePoints
{
  MakePoints(double x, double y) { operator()(x,y); }

  MakePoints& operator()(double x, double y) { m_data.push_back(Point(x,y)); return *this;}

  operator std::vector<Point>() const { return m_data;}

  std::vector<Point> m_data;
};
...
EXPECT_EQ(35.0, PerimeterLength(MakePoints(5,10)(5,20)(15,20)(15,5)));
```

with messages more like the ideal:

```
EXPECT_EQ failed: PerimeterLength(MakePoints(5,10)(5,20)(15,20)(15,5))) (34.0) != 35.0
```

and more generically

```
template< typename T >
struct MakeVectorImpl
{
  template< typename TData >
  MakeVectorImpl& operator()(const TData& t) { m_data.push_back(t); return *this;}

  operator std::vector<T>() const { return m_data;}

  std::vector<T> m_data;
};

template< typename T >
MakeVectorImpl<T> MakeVector(const T& t)
{
  return MakeVectorImpl<T>()(t);
}
...

EXPECT_EQ(MakeVector(1)(1)(3)(4)(5)(9)), Sort(MakeVector(3)(1)(4)(1)(5)(9)));
EXPECT_EQ("!Helloworld", Join(Sort(MakeVector<std::string>("world")("Hello")("!"))));
```

No use of macros here, but it got me thinking above improving Logging..

Super LOG macro

Could I log out variable number of different objects, with names?

A bit of experimenting and a `Dr. Dobb's Journal` article (2003) led to:

```
struct Log
{
  Log& LOG_A; Log& LOG_B; ofstream m_file;

  static Log& Instance() { static Log log; return log;}
  Log(): LOG_A(*this), LOG_B(*this), m_file("c:/temp/log.log") {}

  template< typename T >
  Log& Print(char const * name, const T& value)
  { m_file << name << ":" << value << "\t"; return *this;}

  Log& NewLine() { m_file << "\n"; return *this;}
};

#define LOG Log::Instance().NewLine().LOG_A
#define LOG_A(x) Print(#x, (x)).LOG_B
#define LOG_B(x) Print(#x, (x)).LOG_A
```

which allowed:

```
const int a = 5;
const int limit = 10;
string text{"WORLD"};
Point pnt(31.4,1.52);

LOG(a)(limit)("hello")(text)(pnt);
```

to output:

```
a:5 limit:10 "hello":hello text:WORLD pnt:[31.4000,1.5200]
```

by progressively expanding

```
LOG(a)(limit)("hello")(text)(pnt);
Log::Instance().NewLine().LOG_A(a)(limit)("hello")(text)(pnt);
Log::Instance().NewLine().Print("a",a).LOG_B(limit)("hello")(text)(pnt);
Log::Instance().NewLine().Print("a",a).Print("limit",limit).LOG_A("hello")(text)(pnt);
```

Crucially, this made logging so lightweight to add that developers didn't feel the need to build and keep elaborate logging routines, and could concentrate on debugging.

Smart Assert

The key ideas for the Logging were originally described for Smart Asserts in an article in `Dr. Dobb's Journal`: Enhancing Assertions (Andrei Alexandrescu and John Torjo)

Similar macro/class combinations:

```
#define Assert(expr) (!!(expr)) || SmartAssert(#expr, __FILE__, __LINE__).ASSERT_A
#define ASSERT_A(x) Print(#x,(x)).ASSERT_B
```

allowed us to build macros allowing us to write simple, powerful code such as:

```
Assert(Size(line) == Size(markedSegments)+1) (Size(line))(Size(markedSegments));
Assert(!marker->IsPlaced())(player->Turn()) (markerId)(*marker)(board->DelReason());
```

so that the names and values could be written to the Assert dialog if the Assert fired:

```
Assert Failed (Size(line) == Size(markedSegments)+1) Size(line):5 Size(markedSegments):5
Assert Failed (!marker->IsPlaced()) player->Turn():4 markerId:B12 *marker:{..} ..
```

Tidying up

Standard assert code disappears in release builds.

A little care needs to be taken to *eat* the trailing logging code:

```
#ifdef NDEBUG

const int ASSERT_A = 0;
const int ASSERT_B = 0;

#define Assert(x) ASSERT_A
#define ASSERT_A(x) ASSERT_B
#define ASSERT_B(x) ASSERT_A
```

so that

```
Assert(!marker->IsPlaced())(player->Turn())(markerId)(*marker)(board->DelReason());
```

collapses to `0;`

Simple DSL

There is even an argument for using macros to simply hide boilerplate code to allow the important code to be emphasized and worked with e.g. a simple benchmarking harness:

```
#define CALC(calc) TestCalc(#calc, [](const double x, const double xInner, const double xInnerInv) { calc ; })

bool results [] =
{
  CALC( x ),
  CALC( (x < 0.5) ? x : 0 ),
  CALC( (x > 0.5) ? x : 0 ),
  CALC( (x < xInner) ? x : 0 ),
  CALC( (x > xInner) ? x : 0 ),
  CALC( x + 1.0 ),
  CALC( x / xInner ),
  CALC( x * xInnerInv ),
  CALC( x/2 ),
  CALC( x/2.0 ),
  CALC( x*0.5 ),
...
```

so long as the code in the macro is as simple, and stable and unlikely to need debugging. In this case the work is quickly handed to a real function.

Future = Reflection?

Being able to work programmatically with the structure of the program sounds like we should be able to get at string names for variables, types and functions, and possibly build new symbols.

The current state is conveniently summed up in https://meetingcpp.com/blog/items/reflections-on-the-reflection-proposals.html

- Qt uses the moc framework for reflection
- Boost has the fusion and hana libraries
- C++ std proposal on static reflection (2014) is still progressing

Some proposals are more *templatey* than others e.g:

```
template <typename T>
T min(const T& a, const T& b) {
  log() << "min<"
        << get_display_name_v<$reflect(T)>
        << ">(" << a << ", " << b << ") = ";
  return a < b ? a : b;
}
```

vs

```
template<typename T> min(T a, T b) {
  log() << "min" << '<'
    << $T.qualified_name() << ">("
    << $a.name() << ':' << $a.type().name() << ','
    << $b.name() << ':' << $b.type().name() << ") = ";
  return = a < b ? a : b;
}
```

Challenges

1. Create an enum and build one or both mapping functions e.g.

```
START_ENUM(Shape);
ENUM_VALUE(circle);
ENUM_VALUE(square);
FINISH_ENUM;

Shape shape = from_string("circle");
string square_ = to_string(Shape::square);
```

2. Using variadic marcos and functions to allow logging of a list of variables of different types e.g.

```
const int a = 5;
const int limit = 10;
string text{"WORLD"};
Point pnt(31.4,1.52);

LOG(a, limit, text, pnt);
```

to output

```
a:5 limit:10 "hello":hello text:WORLD pnt:[31.4000,1.5200]
```

Any ideas/questions: richard@shepherd.ws