Université Paris I – Panthéon-Sorbonne
90, rue de Tolbiac – 75634 Paris Cedex 13

Masters MAEF, QEM, DU MMEF — June 2016
Resit exam

Introduction to Modern C++

Solutions

Question 1. We can use the following:

```
const auto alpha = 1./3;
const auto beta  = 0.95;
```

The C++ compiler should infer the correct type (e.g., `double`) of these variables at compile time. Note that we use `1.` in the definition of `alpha`: This informs the compiler that the number `1` is to be understood as a floating-point number. Otherwise, the division is interpreted as happening between integers and we would get `alpha = 0`.

Question 2. The following line answers the question:

```
const std::vector<double> z {0.9792, 0.9896, 1.0000, 1.0106, 1.0212};
```

We must import the `vector` library, which is part of the C++ standard library.

Question 3. We have, by definition:

$$\begin{aligned}
\texttt{capitalSteadyState} &= (\alpha\beta)^{\frac{1}{1-\alpha}} \\
\texttt{outputSteadyState} &= (\alpha\beta)^{\frac{\alpha}{1-\alpha}} \\
\texttt{consumptionSteadyState} &= (\alpha\beta)^{\frac{\alpha}{1-\alpha}} - (\alpha\beta)^{\frac{1}{1-\alpha}} \\
&= (1-\alpha\beta)(\alpha\beta)^{\frac{\alpha}{1-\alpha}}
\end{aligned}$$

Again, the difference is that `1.` is a floating-point value, while `1` is an integer. However, here we should expect implicit type conversion to take place, since the type of `alpha` is already known to the compiler.

Question 4. Here is the summary:

Situation	Problem	Type	Result
$\alpha = 1$	Float division by zero	Fatal error	Crash
$\alpha < 0$ and $\beta > 0$	Root of a negative number	Silent error	$\pm$`NaN`
$(\alpha < \frac{1}{2}$ or $\alpha > 1)$ and $\beta \leq 0$	Root of a negative number	Silent error	$\pm$`NaN`
$\alpha \approx 1$ and $\beta \gg 1/\alpha$	Float overflow	Silent error	`inf`
$\alpha\beta$ too large	Float overflow	Silent error	`inf`
α or $\beta =$ `NaN` or `inf`	Propagation of exceptions	Expected behaviour	`NaN` or `inf`

Of these errors, only the first one will cause a crash. But it makes sens to catch as many errors as possible, as early as possible.

Question 5. For the sake of clarity, we create several `if` clauses:

```
if ((alpha == 1)||(alpha!=alpha)||(alpha==inf)) {
  std::cerr << "Alpha should be different from 1, NaN, and inf." << std::endl;
  return;
}
if ((beta!=beta)||(beta==inf)) {
  std::cerr << "Beta should be different from NaN and inf." << std::endl;
  return;
}
if (((beta<=0)&&((alpha>1)||(alpha<0.5)))||((alpha<0)&&(beta>0))){
  std::cerr << "Alpha and beta values are incompatible." << std::endl;
  return;
}
if ((pow(alpha*beta,alpha/(1-alpha))==inf)||(alpha*beta==inf)) {
  std::cerr << "Values are too small or too large." << std::endl;
  return;
}
```

Naturally, all these tests could be put in a single `if` clause. Note that order matters, so that for instance the last test will never raise a division by zero error. Note also that we use `cerr`, the standard error channel, and not `cout` which is reserved to (proper) outputs.

Question 6. The visibility scope of methods and fields in a class refers to which class instance, if any, can access that given field or method. More specifically:

`private` The field or method cannot be accessed from outside the class instance.

`public` The field or method can be accessed from outside the class instance.

`protected` The field or method can be accessed from outside the class instance, but only from a class instance that is a child of (i.e., inherits) the current class. To the rest of the program (e.g., parent classes and unrelated classes) the `protected` field or method cannot be accessed.

Question 7. There are several ways to answer the question. The following answer leverages the `algorithm` standard library:

```
std::array<double, 17820> gridCapital;
std::iota(gridCapital.begin(),
          gridCapital.end(),
          0);
std::for_each(gridCapital.begin(),
              gridCapital.end(),
              [](auto &n){ n = 0.5 * capitalSteadyState + 0.00001 * n; });
```

We used the standard library `array` type, which is probably the most appropriate choice in the situation.

Note that if we use `i` directly for the computation, it will implicitly be cast as a `double`. In practice it should be a form of integer, either `int` or `size_t`. The solution proposed above avoids this issue entirely as no iterator appears in the computation.

An alternative, more traditional answer using C-style `for` loops could be the following:

```
std::array<double, 17820> gridCapital;
for(auto i = 0; i < gridCapital.size(); ++i) {
  gridCapital[i] = 0.5 * capitalSteadyState + 0.00001 * i;
}
```

Note that we should use `++i` and not `i++`.

Question 8. We use here a C++ reference, so that the modifications we make on `grid` are operated on the original variable (and not a copy, as would be the case by default).

```
void InitialiseGrid(std::array<double,17820>& grid) {
  // Insert here code from Question 7
}
```

The function is *impure*, as it has side-effects: In particular, it affects a variable that may live out of its scope.

Question 9. Note that we must specify the matrix's dimensions.

```
Matrix<N,M> InitialiseExpectedValue ();
```

Question 10. The challenge here is twofold: On the one hand, one must be careful to always specify the appropriate type (in particular, matrix dimension and `const` specifier), and on the other we need to perform a *deep-copy*, i.e. a copy element by element. There are again several ways to perform a loop through elements; we chose the following approach:

```
Matrix<N,M> const CopyMatrix(const Matrix<N,M>& A) {
    Matrix<N,M> result;
    auto i = 0;
    for (const auto& row : A) {
      auto j = 0;
      for (const auto& val : row) {
        result[i][j] = val;
        ++j;
      }
      ++i;
    }
    return result;
}
```

Our function is `const`-safe: The compiler will guarantee that this function does not modify the internal state of the class it belongs to, and that the argument `A` will not be modified either. By making sure that variables that shouldn't be modified are `const`, we prevent developers from altering their content, thus avoiding mistakes and some security issues.

Question 11.

`g++` We call the compiler, here `g++`, to perform the operation (by default, compilation)

`-o rbc` We specify that the output will be called `rbc`, using the `-o` (stands for "output") flag

`rbc.cpp` We specify that the input is the file `rbc.cpp`, which contains our C++ code.

`-O3` We ask the compiler for the most aggressive level of optimisation

`--std=C++11` We specify that we use C++ in its 2011 standard. Note that this should not be necessary as C++11 is now the default standard used by `gcc`, but older versions of the compiler may still require that option specified.

Question 12.

⇒ Write proper documentation to explain which values of α and β can be used.
While not *technically* necessary for the program to work, it is good practice to document extensively any source code. It is even more important to do so here, as there are values that are forbidden and may cause unintended behaviour or crashes.

⇒ Create a header file and add the code file to the compilation chain.

The header file (extension `.h`) contains the signatures of classes and functions which are implemented in the code file (extension `.cpp`). While all files must be present during compilation, only the code files are specified explicitly to the compilation chain.

Bonus question. Here is a function that answers the question. Documentation follows the JavaDoc format (compatible with `doxygen`). While the function may not terminate (due to the use of recursion and the limited precision of floats), this behaviour is explicitly stated in the documentation; and if the function terminates then the precision is guaranteed to be correct.

In fact, the implementation below (and Newton's algorithm) would work even if `x` is a complex number — provided we change the type of `mySqrt` accordingly, and use the module instead of absolute value in computing the error.

```
#include <cmath>

double const mySqrt(const auto& x, const auto& rk, const auto& precision) {
  /**
  * Computes the square root of x up to a prescribed precision,
  * using Newton's algorithm.
  *   @param x  : The number whose square root we are computing.
  *               Cannot be < 0.
  *   @param rk : Initial guess for the square root (use x if no idea).
  *               Cannot be <= 0.
  *   @param precision : The maximum error allowed.
  *                      Cannot be <= 0.
  *   @returns The square root of x, or -1 if error.
  *
  *  Note: If the precision required is too high, this algorithm may fail
  *        and run forever due to floating-point limitations in terms of
  *        precision.
  */

  auto error = std::fabs(rk*rk - x);
  if (error < precision) {
      // We have reached the desired precision.
      return rk;
  }

  if ((x <= 0) || (rk <= 0) || (precision <= 0)){
      // Illegal parameter
      return -1;
  }

  // Precision is insufficient, we must iterate
  return mySqrt(x, 0.5*(rk + x/rk), precision);
}
```

The choice of `rk`, the initial guess, impacts how long the algorithm will run. A better choice would yield a faster answer. In the case of complex numbers, where the square root is no longer unique, a different choice of `rk` may lead to a different root.

In the picture below, the same method is used to compute the complex cube roots of 1, which are -1, $\rho = e^{i\pi/3}$, and $\overline{\rho} = e^{-i\pi/3}$. Each of the three roots is associated to a colour: -1 is red, ρ is blue, and $\overline{\rho}$ is green. The picture represents which root we get with Newton's algorithm, depending on our initial guess `rk` (in the complex plane). The resulting image is also known as the "Julia set for $z \mapsto z^3 - 1$", and is a fractal object.