Finite element methods in scientific computing

Wolfgang Bangerth, Colorado State University

Lecture 24:

Best practices in programming:

Defensive programming and other ways to avoid bugs

The truth about programming

Good programmers also introduce bugs – but they are better at fixing them!

The truth about programming

Observations about *good* programmers:

- They make fewer syntax errors
  (because they know the programming language well)
- They make just as many logic error as others with equally much experience
- But they have developed practices to find their errors faster.

So: Let's talk about best programming practices!

The goal

Good programmers:

- Know that they make errors
- Observe what *kinds of errors* they make
- Take active precautions to prevent such errors by changing their programming style

This is called *defensive programming.*

Defensive programming

General approach:

- Assume that every possible error can happen:
  - user errors when calling a function or using a program
  - your own errors when writing code

- Guard against it
- Make debugging easy

Note: The fact that every possible error *will* happen is of course Murphy's Law!

Defensive programming

Example 1:

Read a line from stdin and process it

```
void read_input_line ()
{
  char buffer[1000];
  gets (buffer);                // read till the end of the line

  // … do something with the line just read
}
```

Rationale: What could possibly go wrong? Who would be so foolish to enter more than 999 characters into a line of input?

Defensive programming

Example 1:

Read a line from stdin and process it

```
void read_input_line ()
{
   char buffer[1000];
   fgets (buffer, 1000, stdin);  // read the line, but <1000 chars

   // … do something with the line just read
}
```

Rationale: At one point someone will (maybe accidentally) enter more characters in one line. This would be most awkward to debug!

Defensive programming

Example 2:
Query the number of elements in a *hp::FECollection*

```
using byte = std::int8_t;          // a “byte”: values 0...255

template <int dim>
byte hp::FECollection<dim>::size ()
{
  return collection.size();
}
```

Rationale: What could possibly go wrong? Nobody would ever store more than 256 elements in a collection. This would again be a bug that's very hard to debug!

Defensive programming

Example 2:

Query the number of elements in a *hp::FECollection*

```
using byte = std::int8_t;

template <int dim>
byte hp::FECollection<dim>::size ()
{
  assert (collection.size() <= std::numeric_limits<byte>::max());
  return collection.size();
}
```

Rationale: Someone, sometime, will indeed store more than 256 elements in a collection.

Defensive programming

Example 2:

Query the number of elements in a *hp::FECollection*

```
using byte = std::int8_t;

template <int dim>
byte hp::FECollection<dim>::size ()
{
   assert (collection.size() <= std::numeric_limits<byte>::max());
   return collection.size();
}
```

Here: *assert()* checks that a condition is true, and if not aborts the program with an error message.

Defensive programming

Example 3:
Access an element of a vector

```
double Vector::operator[] (unsigned int i)
{
  return values[i];
}
```

Rationale: Of course the caller of this function will only access valid elements!

Defensive programming

Example 3:
Access an element of a vector

```
double Vector::operator[] (unsigned int i)
{
   assert (i < size());
   return values[i];
}
```

Rationale: This function will, at one point, be called with an invalid index!

Defensive programming

Summary of this approach:

- Poor programmers think they are good enough not to make errors...
- ...and that if they do they can always debug things.

- Good programmers *know* that they and others make errors
- Have learned to expect such errors and deal with them
- Have learned that it is *far far far* easier to *expect a bug and catch it* than to have to debug it later
- Write code anticipating later debugging.

- This is a learned behavior, borne from experience.

Defensive programming

A corollary to defensive programming:

When your checks catch a bug,
do not just print an error – abort the program!

- After accessing a non-existent vector element, nothing in your program can still make sense!
- Error messages on the screen can be hard to see
- Aborting the program makes it easy to find the place in a debugger

Defensive programming

A corollary to the corollary:

Use assertions – there can
never be enough assertions

Ways to avoid bugs

Approach 1: Design by contract

- First, write up a *specification* of what a function/class/... should do, including corner cases
- Only then write the code

Rationale:

- We often only have a vague idea of what a function should do at the beginning
- We cannot write a correct function if we don't know what "correct" means
- This leads to re-writes
- Re-writes are common sources of errors

Ways to avoid bugs

Approach 2: Use coding conventions

- Common scheme to name functions/variable/etc
- Common way to do things
- Makes it easier to remember things
- Avoids errors

Example: If we want to do *tria3 = tria1 + tria2* , do we need to write

GridGenerator::merge_triangulations (tria1, tria2, tria3);

or

GridGenerator::merge_triangulations (tria3, tria1, tria2);

?

Answer: The former, because in deal.II output arguments come *after* input arguments (with few exceptions).

Ways to avoid bugs

Approach 3: If something is constant, mark it as *const*

- Helps readers of the code
- Allows the compiler to help you

Example:

```
void compute_error ()
{
  for (cell=....)
     {
        const unsigned int n_neighbors = cell->n_active_neighbors();
        … much other code …
        if (n_neighbors = 0)
           … do something …
        else
           … do something else ...
```

Ways to avoid bugs

Approach 4: Limit the scope of variables

- Helps readers of the code
- Allows the compiler to help you

Example:

```
void foo ()
{
   int idx1=0, idx2=0, idx3=1;
   … much code …
   for (; idx2<N; ++idx2)
      sum += element[idx2];
   for (; idx3<N; ++idx2)
      partial_sum[idx3] = partial_sum[idx3-1] + element[idx2];
```

Note: Two definite, two possible bugs here.

Ways to avoid bugs

Approach 4: Limit the scope of variables

- Helps readers of the code
- Allows the compiler to help you

Example (much better style):

```
void foo ()
{
   … much code …
   for (int idx2=0; idx2<N; ++idx2)
      sum += element[idx2];
   for (int idx3=1; idx3<N; ++idx2)
      partial_sum[idx3] = partial_sum[idx3-1] + element[idx2];
```

Note: Compiler will find both bugs. Initialization is now safe.

Ways to avoid bugs

Approach 5: Properly indent code

- Helps readers of the code

Example:

```
void foo ()
{
     if (condition1 == true)
          if (condition2 == false && i<14 && j>42)
{
   function_call(copied from somewhere else);
   some->more(calls);
}
   else
                    other_stuff (copied(again.from.somewhere));
```

Note: That's why IDEs like Eclipse automatically indent code for you.

Ways to avoid bugs

Approach 6: Listen to the compiler

- Take warnings seriously
- Fix your code whenever you get a warning

Note: Not all warnings indicate bad thing will happen. But some do – don't ignore them because you can't see them between other warnings!

Example: deal.II's is written for *zero* warnings!

Ways to avoid bugs

Approach 7: Test while you write the code

- Don't wait till you need to put it all together
- Write small programs that test each function (unit tests)

Note: This requires a *test suite*. But you will learn to appreciate having one.

Example: deal.II's testsuite runs ~13,000 tests after every change!

Ways to deal with bugs

Approach 8: Pick up the trash!

- See something do something!
- If you see a missing ‘const’, add it
- If you are wondering about whether a condition is always true, add an ‘assert’
- ...

This is an investment into your code’s future and you own future productivity.

You also want to be proud of your code: Strive for it to be the best version you can produce!

Ways to deal with bugs

Approach 9: Learn to use the appropriate tools!

- Integrated development environments (Eclipse, VS Code, CLion, ...)
- Gdb/Eclipse debugger
- Valgrind's memcheck
- Valgrind's massif/cachegrind/callgrind
- Intel VTune
- …

Note: Some of these are covered in other lectures.

Finite element methods in scientific computing

Wolfgang Bangerth, Colorado State University