Why Your Compiler Hates You (and why it doesn’t really!)

“The Angry Penguin“, used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Warwick RSE

28/09/2023

Compiler Errors

Compiler Errors

- Compiler errors can be hard to understand at the best of times
  - The compiler knows why what you wrote doesn't work, but you can have made a simple mistake that is invalid for complex reasons
- C++ makes the whole thing worse because with templated code the error can be in code that **you haven't written**
- If when the compiler tries to instantiate template code it finds that it just can't it will try and tell you what the problem is, but sometimes the problem isn't clear until you (mentally) work out what would happen for a given type

Compiler Errors

- The general approach to errors is
  - Start at the beginning
    - Later errors may be caused by earlier errors (i.e. failing to create a variable and later errors due to that variable not existing)
  - Read to the end of a given error!
    - Quite often the compiler puts what has actually gone wrong at the end of a given error
  - Read the extra information
    - Many errors, particularly with templated or overloaded functions, give information about what the compiler has tried to do to make compilation work

Non existent types

```
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

File where error was encountered

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existant types

```cpp
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

Function where error was encountered

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

Line and character position of error

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                               ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```cpp
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

Information about error

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```cpp
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

Source code at error location with error highlighted

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

Another error, here caused by the first error - class does not exist so it can’t be a template argument

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Non existent types

```
#include <vector>

int main(){
  std::vector<nonexistent_class> myvec;
}
```

And another knock on error. Template argument 1 is bad, so template argument 2 is as well

```
test.cpp: In function ‘int main()’:
test.cpp:4:15: error: ‘nonexistent_class’ was not declared in this scope
    4 |   std::vector<nonexistent_class> myvec;
      |               ^~~~~~~~~~~~~~~~~
test.cpp:4:32: error: template argument 1 is invalid
    4 |   std::vector<nonexistent_class> myvec;
      |                                ^
test.cpp:4:32: error: template argument 2 is invalid
```

Compilers may vary

```
non_existent.cpp:4:15: error: use of undeclared identifier 'nonexistent_class'
  std::vector<nonexistent_class> myvec;
              ^
1 error generated.
```

- This output is from clang++ on a Mac rather than g++ on Linux
  - We'll only show g++ here, but error message reading is more about the approach than the details
- Some differences, notably doesn't say the function that the error was in (although the line number is still there)
- Also, no knock on errors here - this is unusually helpful for a C++ compiler
- What about a more complex example?

What's wrong with this code?

```
#include <vector>

class demo{
demo(){}
};

int main(){
  std::vector<demo> myvec;
  myvec.resize(10);
}
```

What’s wrong with this code?

```
In file included from /usr/include/c++/11/vector:65,
                 from test.cpp:1:
/usr/include/c++/11/bits/stl_construct.h: In instantiation of ‘void std::_Construct(_Tp*, _Args&& ...) [with _Tp = demo;
_Args = {}]’:
/usr/include/c++/11/bits/stl_uninitialized.h:579:18:   required from ‘static _ForwardIterator
std::__uninitialized_default_n_1<_TrivialValueType>::__uninit_default_n(_ForwardIterator, _Size) [with _ForwardIterator =
demo*; _Size = long unsigned int; bool _TrivialValueType = false]’
/usr/include/c++/11/bits/stl_uninitialized.h:640:20:   required from ‘_ForwardIterator
std::__uninitialized_default_n(_ForwardIterator, _Size) [with _ForwardIterator = demo*; _Size = long unsigned int]’
/usr/include/c++/11/bits/stl_uninitialized.h:704:44:   required from ‘_ForwardIterator
std::__uninitialized_default_n_a(_ForwardIterator, _Size, std::allocator<_Tp>&) [with _ForwardIterator = demo*; _Size =
long unsigned int; _Tp = demo]’
/usr/include/c++/11/bits/vector.tcc:627:35:   required from ‘void std::vector<_Tp,
_Alloc>::_M_default_append(std::vector<_Tp, _Alloc>::size_type) [with _Tp = demo; _Alloc = std::allocator<demo>;
std::vector<_Tp, _Alloc>::size_type = long unsigned int]’
/usr/include/c++/11/bits/stl_vector.h:940:4:   required from ‘void std::vector<_Tp, _Alloc>::resize(std::vector<_Tp,
_Alloc>::size_type) [with _Tp = demo; _Alloc = std::allocator<demo>; std::vector<_Tp, _Alloc>::size_type = long unsigned
int]’
test.cpp:9:15:   required from here
/usr/include/c++/11/bits/stl_construct.h:119:7: error: ‘demo::demo()’ is private within this context
  119 |       ::new((void*)__p) _Tp(std::forward<_Args>(__args)...);
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.cpp:4:1: note: declared private here
    4 | demo(){}
      | ^~~~
```

What's wrong with this code?

- Well that's unhelpful
- Remember that for templated code like **std::vector** the compiler is only generating the code for the vector for your type when you tell it to create the vector
  - Actually, if you check the line number in the error there isn't a problem until you try to reserve the items, but why is quite complicated
- If you read through to the bottom of the error (where the line number is), it tells you the problem

What's wrong with this code?

```
/usr/include/c++/11/bits/stl_construct.h:119:7: error: ‘demo::demo()’ is private within this context
  119 |       ::new((void*)__p) _Tp(std::forward<_Args>(__args)...);
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.cpp:4:1: note: declared private here
    4 | demo(){}
      | ^~~~
```

- The problem is that when I defined the **demo** class I forgot that members are private by default, so the constructor function for it is private
- Privacy does apply to constructors, so if the constructor is private you can't build the object (there are reasons why you'd want to do this)
- Simply put in a **public:** directive or make **demo** a **struct** rather than a **class**

What does this show?

- This shows the major problem with templated code
- The problem isn't directly in your code, it is in the templated code when it tries to create that code with the type that you instantiated it with
- So you see errors from the templated code
  - For **std::vector** that is almost never what you want - you want to know what **you** did wrong when creating the vector
  - For your own templated code, there might be errors in it, so the compiler has to show you
- **Your compiler really is doing the best that it can!**

Beyond the Compiler

What next?

- Your code compiling does not indicate that your job is done
  - Successfully compiling code is **syntactically correct** - follows the rules of grammar for C++
  - “Love is yellow” is a syntactically correct English sentence, but would generally be considered meaningless
- Even if your code compiles it can be incorrect
  - Completely incorrect (something is always invalid, but the compiler can’t tell)
  - Conditionally incorrect (mostly something is correct, but under some circumstances it will be invalid)

Undefined behaviour

- Most languages have the idea of **undefined behaviour**
- That is code that is syntactically **valid** but is not guaranteed to have any given result
- Mostly it is fairly obvious things like accessing vectors with an index outside the bounds of the vector (there is a way of accessing vectors with bounds checking, but that slows the code down so isn't normally used in production code)
- Sometimes there are unexpected examples of undefined behaviour

Undefined Behaviour

```
#include <iostream>

int main(){
  int x = 5;
  x = x++ + x++;
  std::cout << x << "\n";
}
```

- In this example, you are modifying the value of **x** in different ways in a single line and C++ actually doesn't define what has to happen
- You might expect the answer to be 6+7=13
  - i.e increment x once (5+1=6), then increment it again (6+1)=7 and then add the results
- None of the compilers that we tested give that answer
  - gcc and clang give 11 and Intel's classic C++ compiler gives 12

Undefined behaviour

- To be clear, all three compilers **are correct**
- It isn't what you would expect but that C++ code invokes **undefined behaviour** and the compiler is permitted to create code that gives any result at all
  - Technically including formatting your hard drive and applying for another job in your name that doesn't involve working with computers, but that is unlikely
- Helpfully, while the compilers will **compile** this code, if you turn on **warnings** then they will often be able to tell you if you are making mistakes that are not **grammatical mistakes** with the language

Undefined Behaviour

```
test.cpp:5:5: warning: operation on ‘x’ may be undefined [-Wsequence-point]
    5 |    x = x++ + x++;
      |    ~~^~~~~~~~~~
test.cpp:5:5: warning: operation on ‘x’ may be undefined [-Wsequence-point]
test.cpp:5:5: warning: operation on ‘x’ may be undefined [-Wsequence-point]
test.cpp:5:5: warning: operation on ‘x’ may be undefined [-Wsequence-point]
```

- The compiler flags for turning on warnings with compilers are compiler specific
- With g++(gcc) that we’ve been using use **-Wall -Wextra**
- **Without those flags g++ doesn’t say anything when you compile that code**
- Clang actually does even without flags, but don’t rely on this!

Optimisation

- A classic symptom of undefined behaviour is that the result will change due to unrelated changes
  - Changing when you change optimisation level (i.e. add **-O3** as a compiler flag for maximum optimisation) is a classic example
- It can reorder operations, remove redundant operations and many other things
- Modern optimising compilers technically only guarantee to produce a program with the same effects as the one you wrote (usually called **as if** compilation)
  - This is why undefined behaviour causes chaos - your code isn't then required to have any particular effects so things change in response to anything
  - Undefined behaviour anywhere in a program makes the behaviour of the entire program undefined

Other errors

- Compiler warnings are really useful for identifying programming problems that are visible from the code that you have written
  - There are more powerful tools for this type of analysis called **static analysers** (**cppcheck** is a good one for C++)
- They can't find problems that are only apparent when the code is running
  - This problem is generally called **dynamic analysis**
- Static analysis just looks at your code "on the page", dynamic analysis looks at the actual running code

Debuggers

- The most common type of dynamic analysis tool is a debugger
- Debuggers allow you to interrupt a code as it is running and look at the state of variables etc. They can also automatically interrupt a code as it is crashing so you can see what variables might be in a state to cause the crash
- **gdb** that comes with **gcc** is free and powerful. There are lots of tutorial on gdb, so we won't cover it
- **valgrind** is a suite of tools that can evaluate more problems than gdb, including finding memory leaks and use of uninitialised memory

Profilers

- Sometimes the problem isn't that your code is crashing or giving the wrong answer, it is just too slow
- In this case you want a **profiler**
- Generally profilers such as **gprof** will tell you which parts of your code are taking the most time (normally at the function level, but you can do line by line profiling) so that you know where to speed up
- You still have to write faster code!
- There are also profilers that can inspect the state of your CPU or GPU to find out if your performance problem is with you not using the right bits of your processor

Final notes

- Just because your code compiles and runs without crashing doesn't mean that it works
- Tools like **valgrind** will let you know about things like unintialised variables which will make your results nonsense
- Sometimes the problem isn't in the code at all - you either haven't coded up what you think you have or your approach fundamentally isn't right for the problem that you are trying to solve
- **Test your code for correctness**
  - How you test is far, far less important than that you **do** test
  - There are automatic systems, but they are only as much help as you make them