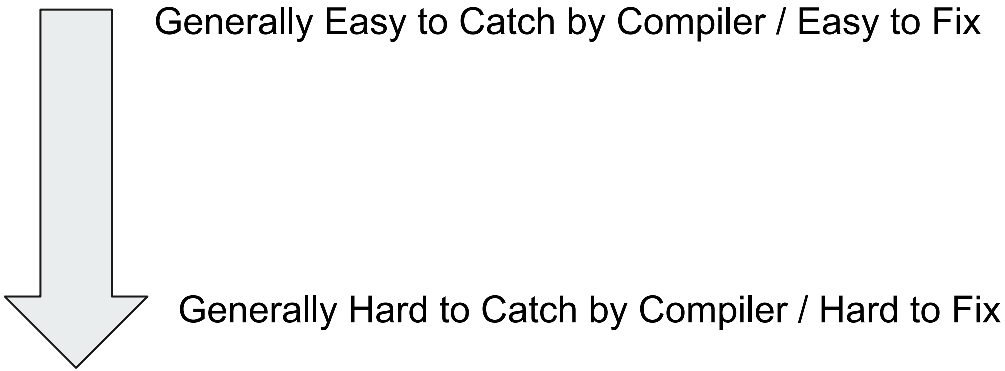
Bad C++ Code

Las Vegas C/C++ Meetup Group - Tuesday October 9th, 2018

Presented by : Ray Imber

Types of Bad C++

- Syntax Errors
- Semantic Errors
- Logic Errors
- Code Design Errors



Generally Easy to Catch by Compiler / Easy to Fix

Generally Hard to Catch by Compiler / Hard to Fix

Syntax Errors

Common Syntax Errors

- **Missing semicolons**

```
a = x+y
b = m/n;
```

Common Syntax Errors

- **Missing comment tags**

```
/* comment line 1
Statement1;
Statement2;
/* comment line 2 */
Statement 3;
```

Common Syntax Errors

- **Misplaced semicolons**
- **Compiler won’t catch this one!**

```
for(i =1; i<=10; i++);
     sum = sum + i;
```

Common Syntax Errors

- **Missing Braces**
- **Compiler won’t catch this either**

```
for(i=1;i<=10;i++)
     sum1=sum1+i;
     sum2=sum2+i*i;
printf(“%d%dn”,sum1,sum2);
```

Common Syntax Errors

- **Precedence of Operators**
- **Compiler won’t catch this one**

```
if(value = product() >= 100)
            tax=0.05*value;
```

Common Syntax Errors

- **Omitting Parentheses around Arguments in Macro Definitions**

```
#define f(x) x*x+1
y=f(a+b);
```

Common Syntax Errors

- **Omitting Parentheses around Arguments in Macro Definitions**

```
#define f(x) x*x+1
y=f(a+b);
```

```
Evaluates As:
y=a+b * a+b+1;
```

Common Syntax Errors

- **Using assignment for equality comparison**

```
Char done = 'Y';
while (done = 'Y')
{
     cout << "Continue? (Y/N)"; cin >> done;
}
```

Common Syntax Errors

- **Using assignment for equality comparison**

```
Char done = 'Y';
while (done == 'Y')
{
    cout << "Continue? (Y/N)"; cin >> done;
}
```

Semantic Errors

Common Semantic Errors

- **Array Bounds / Off by one**
- **Compiler won’t catch this one**
- **In real world scenarios it’s not as obvious (array may be passed in as a parameter or dynamic)**

```
int x[10],sum,i;
for(i=1;i<=10;i++)
     sum=sum+x[i];
```

Common Semantic Errors

- **Using an Uninitialized Pointer**

```
main()
{
     int a,*ptr;
     a=25;
     *ptr=a+5;
}
```

Common Semantic Errors

- **Missing/Incorrect Pointer Operators**

```
main()
{
    int m;
    int *p1;
    m=25;
    p1=m;
    printf(“%d\n”,*p1);
}
```

Common Semantic Errors

- **Incorrect Arguments Passed to Scanf()**

```
main()
{
     int code;
     scanf(“%d”, code);
}
```

Common Semantic Errors

- **Incorrect Arguments Passed to Scanf()**
- **Scanf() takes pointers!**

```
main()
{
    int code;
    scanf(“%d”, &code);
}
```

Common Semantic Errors

- **Switch Statements and Break Statements**

```
int x = 2;
switch(x)
{
    case 2:
        cout << "two" << endl;
    case 3:
        cout << "three" << endl;
}
```

Common Semantic Errors

- **Switch Statements and Break Statements**

```
int x = 2;
switch(x)
{
    case 2:
        cout << "two" << endl;
        break;
    case 3:
        cout << "three" << endl;
        break;
}
```

Common Semantic Errors

- **Writing to a string literal**

```
main() {
    char * c = "hello";
    *c = 'B';
}
```

Common Semantic Errors

- **Writing to a string literal**

```
main() {
    char c[] = "hello";
    *c = 'B';
}
```

Common Semantic Errors

- **delete[] on new or delete on new[]**

```
main() {
     Foo *bar = new Foo();
     Foo *bars = new Foo[100];
     //….
     delete[] bar;
     delete bars;
}
```

Common Semantic Errors

- delete[] on new or delete on new[]
- Don’t use new or delete. Use smart pointers / std data structures instead
- Std::uniqe_ptr
- std::Vector

Common Semantic Errors

- **Most vexing parse**

```
class Timer {
 public:
  Timer();
};

class TimeKeeper {
 public:
  TimeKeeper(const Timer& t);

  int get_time();
};

TimeKeeper time_keeper(Timer());
```

Is it:

- a variable definition for variable time_keeper of class TimeKeeper, initialized with an anonymous instance of class Timer

Or

- a function declaration for a function time_keeper that returns an object of type TimeKeeper and has a single (unnamed) parameter that is a pointer to function returning type Timer (and taking no input).

Common Semantic Errors

- **Most vexing parse** https://en.wikipedia.org/wiki/Most_vexing_parse

```
class Timer {
 public:
  Timer();
};

class TimeKeeper {
 public:
  TimeKeeper(const Timer& t);

  int get_time();
};

TimeKeeper time_keeper(Timer());
```

Believe it or not, the C++ standard requires it to be the second option:

- **a function declaration for a function time_keeper that returns an object of type TimeKeeper and has a single (unnamed) parameter that is a pointer to function returning type Timer (and taking no input).**

Common Semantic Errors

- **Most vexing parse**

Can be fixed by extra parenthesis:

```
TimeKeeper time_keeper( (Timer()) );
```

Or

Variable Initialization

```
TimeKeeper time_keeper = TimeKeeper(Timer());
```

Logic Errors

Common Logic Errors

```
void SomeMethod()
{
  ClassA *a = new ClassA;
  SomeOtherMethod();
  delete a;
}
```

Common Logic Errors

```
void SomeMethod()
{
  ClassA *a = new ClassA;
  SomeOtherMethod();
  delete a;
}
```

Can throw an exception!

Common Logic Errors

```
void SomeMethod()
{
  std::unique_ptr<ClassA> a(new ClassA);
  SomeOtherMethod();
}
```

Can still throw an exception,
But **uniqe_ptr** will correctly call the
Destructor if it does.

Side Note: RAII

Resource acquisition is initialization

It’s about Destruction as well as Creation of Objects

Applies to Pointers, but also other resources such as network or file handles

Smart Pointers are the perfect example of RAII

The technique was developed for exception-safe resource management in C++ primarily by Bjarne Stroustrup and Andrew Koenig

https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Common Logic Errors

```
class MyString : public std::string
{
     ~MyString() {
     // ...
     }
};

int main()
{
     std::string *s = new MyString();
     delete s;
}
```

Virtual Destructor

Common Logic Errors

```
class MyString : public std::string
{
     ~MyString() {
     // ...
     }
};

int main()
{
     std::string *s = new MyString();
     delete s;
}
```

Won’t call the custom virtual destructor b/c it thinks it’s a std::string not a MyString!

Common Logic Errors

Where is the Memory leak?

```
void func(void)
{
    std::unique_ptr<int> my_array(new int[5]);
}

int main()
{
    func();
    Return 0;
}
```

Common Logic Errors

A smart pointer will call a delete operator without [] brackets if you don’t tell it that you are pointing to an array!

```
void func(void)
{
    std::unique_ptr<int[]> my_array(new int[5]);
}

int main()
{
    func();
    Return 0;
}
```

Common Logic Errors

What’s wrong with this code?

```
Complex& SumComplex(const Complex& a, const Complex& b)
{
    Complex result;
    …..
    return result;
}

Complex& sum = SumComplex(a, b);
```

Common Logic Errors

```
Complex& SumComplex(const Complex& a, const Complex& b)
{
    Complex result;
    …. .
    return result;
}

Complex& sum = SumComplex(a, b);
```

Result is a local variable, created on the stack. It will go away as soon as the function returns!

Common Logic Errors

```
Complex SumComplex(const Complex& a, const Complex& b)
{
     return Complex(a.real + b.real, a.imaginar + b.imaginar);
}

Complex sum = SumComplex(a, b);
```

For most of today’s compilers, if a return line contains a constructor of an object the code will be optimized to avoid all unnecessary copying - the constructor will be executed directly on the “sum” object.

Common Logic Errors

What happens with this code?

```
class A
{
public:
   A(){}
   ~A()
   {
      writeToLog();
   }
};
```

```
try
{
   A a1;
   A a2;
}
catch (std::exception& e)
{
   std::cout << "exception caught";
}
```

writeToLog() throws an exception

Common Logic Errors

two exceptions in parallel, no matter whether they are of the same type or different type the C++ runtime environment does not know how to handle it and will terminate!

```
class A
{
public:
   A(){}
   ~A()
   {
      writeToLog();
   }
};
```

```
try
{
   A a1;
   A a2;
}
catch (std::exception& e)
{
   std::cout << "exception caught";
}
```

The catch is never called!

Common Logic Errors

What’s wrong with this code?

```
vector<string> v;
v.push_back(“string1”);
string& s1 = v[0]; // reference
vector<string>::iterator iter = v.begin(); // iterator
v.push_back(“string2”);
cout << s1;     // access to a reference
cout << *iter;  // access to an iterator
```

Common Logic Errors

```
vector<string> v;
v.push_back(“string1”);
string& s1 = v[0]; // reference
vector<string>::iterator iter = v.begin(); // iterator
v.push_back(“string2”);
cout << s1;      // access to a reference
cout << *iter;   // access to an iterator
```

Could cause the vector to resize,
Making both the reference and the
Iterator invalid.

This is more common in multithreaded code

Common Logic Errors

What’s wrong with this code?

```
#include <iostream>
#include <fstream>
int main()
{
  std::ifstream in("input.txt");
  if (!in.is_open())
  {
    std::cerr << "Failed to open file\n";
    return 1;
  }

  int i, j, k;
  in >> i >> j >> k;
  std::cout << calculate(i, j, k);
}
```

Common Logic Errors

always check your I/O operations!

```
#include <iostream>
#include <fstream>
int main()
{
  std::ifstream in("input.txt");
  if (!in.is_open())
  {
    std::cerr << "Failed to open file\n";
    return 1;
  }

  int i, j, k;
  in >> i >> j >> k;
  std::cout << calculate(i, j, k);
  return 0;
}
```

What happens if the read operation failed?
I,j, and k are just garbage

Common Logic Errors

always check your I/O operations!

```
int i, j, k;

if (in >> i >> j >> k)
{
  std::cout << calculate(i, j, k);
}
else
{
  std::cerr <<
    "Failed to read values from the
file!\n";
  throw std::runtime_error("Invalid input
file");
}
```

Common Logic Errors

What’s wrong with this code?

```
class A
{
public:
   virtual std::string GetName() const {return "A";}
    …
};

class B: public A
{
public:
   virtual std::string GetName() const {return "B";}
   ...
};
```

```
void func1(A a)
{
   std::string name = a.GetName();
   ...
}

B b;
func1(b);
```

Common Logic Errors

The slicing problem:

This code will compile. Calling of the “func1” function will create a partial copy of the object “b”, i.e. it will copy only class “A”’s part of the object “b” to the object “a”.

```
class A
{
public:
   virtual std::string GetName() const {return "A";}
    …
};

class B: public A
{
public:
   virtual std::string GetName() const {return "B";}
   ...
};
```

```
void func1(&A a)
{
   std::string name = a.GetName();
   ...
}

B b;
func1(b);
```

Common Logic Errors

The Slicing Problem is common when handling exceptions!

```
class ExceptionA: public std::exception;
class ExceptionB: public ExceptionA;

try
{
   func2();
}
catch (ExceptionA ex)
{
    writeToLog(ex.GetDescription());
    throw;
}
```

Throws an ExceptionB exception

Common Logic Errors

- **What’s wrong with this code?**

```
using request_id  = uint32_t;
using receiver_id = uint32_t;

token remove(request_id req, receiver_id rec);

token init_remove(receiver_id receiver)
{
 auto req = new_request();
 return remove(receiver, req);
}
```

Common Logic Errors

- **What’s wrong with this code?**

```
using request_id  = uint32_t;
using receiver_id = uint32_t;

token remove(request_id req, receiver_id rec);

token init_remove(receiver_id receiver)
{
 auto req = new_request();
 return remove(receiver, req);
}
```

The arguments are swapped,
But they are both ints!
No compiler help!

Common Logic Errors

Strongly Typed C++ / Named Types / Tagged Types

- https://vimeo.com/292931307
- don't use naked ints. Use distinct types to signal intent (units of measure / handles / etc...)
- don't use magic numbers
- times and durations in std::chrono are the perfect example
- Libraries to help with this:
- https://github.com/joboccara/NamedType

Code Design Errors

“Code Smell”

- popularised by Kent Beck on WardsWiki in the late 1990s
- certain structures in the code that indicate violation of fundamental design principles or negatively impact code
- Not technically bugs (but can be)
- https://en.wikipedia.org/wiki/Code_smell

Examples of Code Smell

- Duplicated code: identical or very similar code exists in more than one location.
- Contrived complexity: forced usage of over complicated design patterns where simpler design would suffice.
- Shotgun surgery : a single change needs to be applied to multiple classes at the same time.
- Too many parameters to a function
- Excessively long line of code
- Excessively Long method or Class
- Cyclomatic complexity: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
- Many more....

Good C++

https://cpppatterns.com/

Good Resource for common C++ idioms and patterns
with detailed explanations.

https://isocpp.org/faq

The official ISO C++ FAQ
Detailed explanations of common C++ questions written by the
Very people in charge of C++

Resources

- https://isocpp.org/faq
- https://cpppatterns.com/
- http://wiki.c2.com/?ClassicOoAntiPatterns
- https://www.toptal.com/c-plus-plus/top-10-common-c-plus-plus-developer-mistakes
- https://www.learncpp.com/cpp-programming/eight-c-programming-mistakes-the-compiler-wont-catch/
- https://en.wikipedia.org/wiki/Most_vexing_parse
- https://www.reddit.com/r/cpp/comments/489f9l/open_source_projects_with_examples_of_good_modern/
- https://speakerdeck.com/rollbear/ndc-techtown-type-safe-c-plus-plus-lol
- https://vimeo.com/292931307
- https://www.boost.org/doc/libs/1_68_0/libs/gil/doc/html/index.html
- https://github.com/NVIDIA/cuda-samples
- https://github.com/GPUOpen-Tools