Unexpected behavior in C++

Arno Lepisk

`arno@lepisk.se` / `arno.lepisk@hiq.se`

`@arno_l`

SwedenCpp Stockholm 2019-09-26

About me

Arno Lepisk

Software engineering consultant

Unexpected behavior?

Perfectly legal code which does something unexpected

Not to be confused with undefined behavior!

Three kinds of unexpected behavior

Odd

Murphy

Macciavelli

Do not try this at home!

... well do, but not in production code!

What will print?

```
void print(int num) {
    std::cout << (num / 100)%10
              << (num / 010)%10
              << (num / 001)%10 << std::end
}
...
print(15);
print(111);
```

```
015
131
```

An URL is valid C++ code!

```
void foo() {
  http://isocpp.org
}
```

An URL is valid C++ code?

```
void foo() {
  http://isocpp.org
  http://java.com
       ^^---- ERROR!
}
```

Why?

```
void foo() {
  http://isocpp.org
  ...
  goto http;
}
```

Spaceships pre-C++20

C++20 introduces the spaceship operator `<=>`

But you can make spaceship like stuff in older C++ as well!

C++17 spaceship

```
template<auto...> struct d {};

d<>b

;
```

C++98

```
void foo() {

<%%>

int a  <:0:> ;

<::> <%%> ; // C++11

}
```

What happens?

Digraphs

`{`	`<%`
`}`	`%>`
`[`	`<:`
`]`	`:>`

```
<%%>                {}
int a<:0:>;         int a[0];
<::><%%>;           [][{}];
```

alternative operators

`&&`	`and`
`&`	`bitand`
`||`	`or`
`|`	`bitor`
`!`	`not`
`~`	`compl`

Alternative operators

```
int i;
int * ptr = &i;
int * ptr = bitand i;
```

Alternative operators 2

```
class C {
  C(const C bitand);
  C(C and);
  compl C();
}
```

```
class C {
  C(const C &);
  C(C &&);
  ~C();
}
```

Quiz time!

```cpp
#include <iostream>
int main() {
   int i = 1;
   // What will be printed??/
   while(i--)
   {
      std::cout << i << '\n';
   }
}
```

```
C++17:                      pre-C++17:
0                           1
```

Wait! what?

`??/` is a tri-graph, translated to `\`

```cpp
#include <iostream>
int main() {
  int i = 1;
  // What will be printed\
  while(i--)
  {
    std::cout << i << '\n';
  }
}
```

Which overload is called?

```
void foo(char);
void foo(int);
```

```
char c = 'A';
foo(c);    // foo(char)
foo(c+1); // foo(int)
foo(c+c); // foo(int)
```

Moving into murphy territory!

```
c |= 1;
foo(c); // call foo(char)
```

Refactor to

```
auto c2 = c | 1;
foo(c2); // call foo(int)
```

Integral promotion

Arithmetic operations on `char`s are converting the `char`s to `ints`*

```
char + int  -> int
char + char -> int
```

Actual seen usage

```
void print(char c) {
  std::cout << +c << '\n';
}
```

```
void print(char c) {
  std::cout << static_cast<int>(c) << '\n';
}
```

The devil

```
#include <algorithm>
#include <vector>
using namespace std;
int main() {
    std::vector<int> v{7,2,1,3,5,8};
    sort(rbegin(v), rend(v));
}
```

Compilation Ok??

Argument Dependent Lookup (ADL)

So, what is happening here?

```
std::vector<int> v{7,2,1,3,5,8};
sort(rbegin(v), rend(v));
```

Looks for `rbegin` in `std`, since the type of the argument (`std::vector`) is in `std`

etc

Compilation problems

```
namespace ns {
struct B {};
void foo(const B &);
}

void foo(const ns::B &);

int main() {
  ns::B b;
  foo(b); // ERROR
}
```

Foot-gun

```
namespace ns {
struct B {};
struct C : public B {};
void foo(const C &);
}



void foo(const ns::B &);
```

```
int main() {
   ns::C c;
   foo(c); // calls ns::foo(...)
}
```

Short-circuiting

```
void foo(C * ptr) {
  ptr && ptr->foo();
}
```

```
void ensure_cached() {
  in_cache || calculate_cache();
}
```

Short-circuiting short-circuiting

```
void foo(my_smart_ptr<C> * ptr) {
  ptr && ptr->foo();
}
```

```
template<typename T>
bool operator&&(const my_smart_ptr<T> & ptr
```

```
Potential crash!
```

Avoiding the problem

Add `operator bool()` to `my_smart_ptr`

```
void foo(my_smart_ptr<C> * ptr) {
  ptr && ptr->foo();
  // calls builtin operator&&(bool, bool)
}
```

Problems with punctation

```
void foo(int a, int b=0);

foo(1,2);
foo((1,2)); // calls foo(2,0);
```

more comma confusion

```
int foo() {
    return 1,2; // returns 2
}
```

```
int bar() {
    int i;
    i = 1,2;  // (i = 1),2;
    return i; // returns 1
}
```

Enter Machiavelli

```cpp
struct S {};

const S & operator,(const S & s1, const S &
{ return s1; }
```

```cpp
int operator,(const S & s1, const S & s2)
{ return 0; }
```

Who uses the comma operator anyway?

```
for(Idx i = 0, j = 10; i < j; (void)i++, j-
    ...
}
```

Bonus

```
void foo(int a, int b=0);

foo(1,2);
foo((1,2)); // calls foo(2,0);
foo,(1,2);  // does nothing!
```

What is the address of an object?

```
const char * operator&(const MyClass &a)
{ return "Machiavelli was here!"; }
```

```
void func() {
    MyClass a;
    std::cout << &a << std::endl;
    std::cout << std::addressof(a) << std:::
}
```

```
Machiavelli was here!
0xXXXXXXXX
```

`std::addressof`

Always returns the address of an object

```
MyClass m;

auto ptr1 = &m;                     // ???
auto ptr2 = std::addressof(m); // MyClass*
```

Conclusion

C++ is a complex language

Don't make it worse by surprising users

Don't do unnecessary "clever" stuff - write maintainable code

Arno Lepisk

arno@lepisk.se / arno.lepisk@hiq.se

@arno_l