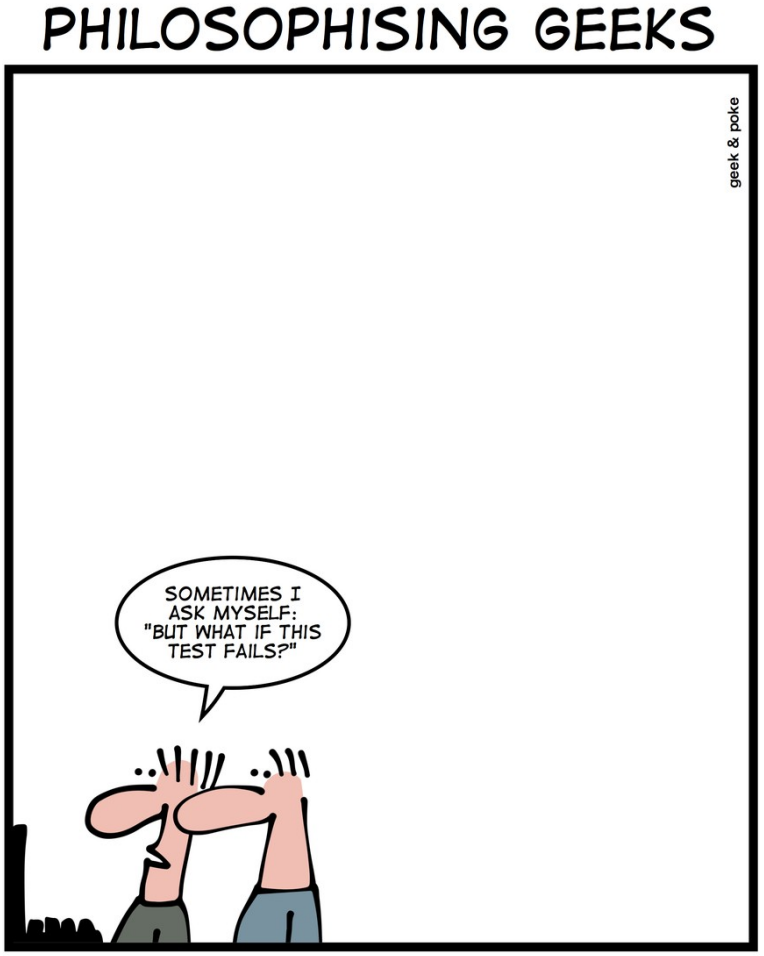
C++ Quiz



Krishna Kumar

Fun with Type Deduction

- const int cx = 0;
- auto cx2 = cx;
- decltype(cx) cx3 = cx;
- template <typename T>

  void f1(T param);

  f1(cx);
- template <typename T>

  void f1(T& param);

  f1(cx);
- template <typename T>

  void f1(T&& param);

  f1(cx);

- Type? Why? //Type is int
- Type? Why? // Const int
- T's type and why?

  param is a copy of cx – int

- T's type and why?

  Referring to a chunk of memory – const int
- T's type and why?

  Neat trick to allow argument forwarding – perfect forwarding

  const int&

Fun with Type Deduction: Solution

- const int cx = 0;
- auto cx2 = cx;
- decltype(cx) cx3 = cx;
- template <typename T>
  void f1(T param);
  f1(cx);
- template <typename T>
  void f1(T& param);
  f1(cx);
- template <typename T>
  void f1(T&& param);
  f1(cx);

- //Type is int
- // Const int
- T's type and why?
- T's type and why?
- T's type and why?

Lambda expressions - Type

- const int cx = 0;
- auto lam = [cx] {cx = 10;};
- // Compiler generated class
- class UptoCompiler {
  - private:
    - ??? cx;
- };

- // What happens here?
- // type? Why?

Lambda expressions: Solution

- const int cx = 0;
- auto lam = [cx] {cx = 10;};
- // Compiler generated class
- class UptoCompiler {
  - private:
    - ??? cx;
- };

- // Error! Why?

- // const int
- The variable cx within {} is what is in the compiler generated class.
- To preserve what is being passed to the hidden compiler generated class.
- Programming in 2 scopes!

Lambda init capture: C++14

- const int cx = 0;
- auto lam = [cx = cx] {cx = 10;};
- // Compiler generated class
- class UptoCompiler {
  - private:

    ??? cx;
  - Public:

    void operator()() const

    { cx = 0; }

    ...
- };

- //Error why?
- // type? Why?
-
-

Lambda init capture: Solution

- const int cx = 0;
- auto lam = [cx = cx] {cx = 10;};
- // Compiler generated class
- class UptoCompiler {
  - private:

    ??? cx;
  - Public:

    void operator()() const

    { cx = 0; }

    ...
- };

- // Error! Why?
- // int (acts like a const int!)
-

Lambda init capture: mutable

- const int cx = 0;
- auto lam = [cx = cx] mutable {cx = 10;};
- auto lam = [cx = cx] ()mutable {cx = 10;};
- class UptoCompiler {
  - private:

    ??? cx;
  - Public:

    void operator()() const

    { cx = 0; }

    ...
- };

- //Error why?
- // Standard failed to add empty() for mutable!
-
- Type??
- Int (acts like an int)! Phew!

Type deduction

For const int cx = 0;

Context	Type
auto	int
decltype	const int
template (T param)	int
template (T& param)	const int
template (T&& param)	const int&
lambda (by-value capture)	const int
lambda (init capture) - same as auto!	int

Type deduction & initialisation

- int x1 = 0;

  int x2(0);

  int x3 = {0};

  int x4 {0};
- auto x1 = 0;

  auto x2(0);

  auto x3 = {0};

  auto x4 {0};
- template<typename T>

  void f(T param)

  f({0});

- // type? Why? - int
- // type? Why? - int
- // type? initializer_list<int>
- // type? initializer_list<int>
- // type? Why?

  Error!

  No type “{0}”

auto to explicit type deduction

```
std::map<std::string, int> m;
```

- // Why this is inefficient

```
for (const std::pair<std::string, int>& p : m) …
```

- // This is optimised

```
for (const auto& p : m) ….
```

auto or explicit type deduction: solution

- Avoid accidental temporary creation!

  std::map<std::string, int> m;
- // Holds object of type std::pair<**const** std::string, int>
- // Why this is inefficient
- // creates a temp on each iteration std::string is copied

  for (const std::pair<std::string, int>& p : m) …
- // This is optimised
- // No temporaries are created

  for (const auto& p : m) ….

References

- Scott Meyers “The last thing D needs”
- Cracking the C, C++ and Java interview