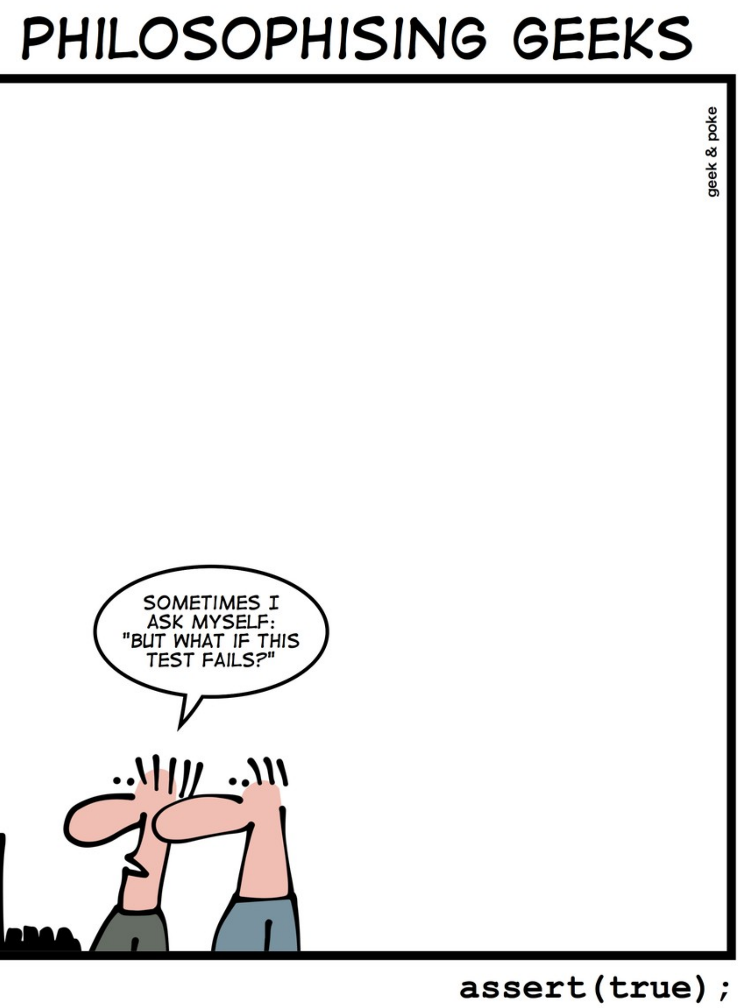
C++ Quiz



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) ….

Inheritance - should this compile?

- class Base {
  - public:

      void dobasework();
- };
- class Derived : public Base{
  - public:

      void derivedwork() {

          dobasework();

      }
- };

- template <typename T>
- class Base {
  - public:

      void dobasework();
- };
- template <typename T>
- class Derived : public Base<T>{
  - public:

      void derivedwork() {

          dobasework();

      }
- };
- Dependent base class
- Two-phase lookup

Inheritance - specilaisation

- template <typename T>
- class Base {
  - public:

        void dobasework();
- };
- template <typename T>
- class Derived : public Base<T>{
  - public:

        void derivedwork() {
            dobasework();
        }
- };
- // cont...

- // cont…
- // Possible future
- // no base work

  template<>

  class Base<int> {};
- // Fail!

  Derived<int> d;

  do.derivedWork();

Inheritance – specilaisation: solution?

- template <typename T>
- class Base {
  - public:

        void dobasework();
- };
- template <typename T>
- class Derived : public Base<T>{
  - public:

        void derivedwork() {
            Base<T>::dobasework();
        }
- };
- // Base<T>:: turns off virtual function calls – not relevant here

- template <typename T>
- class Base {
  - public:

        void dobasework();
- };
- template <typename T>
- class Derived : public Base<T>{
  - public:

        void derivedwork() {
            this->dobasework();
        }
- };
- // Lookup only in the current

Templates & function calls

1) template<typename T1, typename T2> void f(T1, T2);
2) template<typename T> void f(T);
3) template<typename T> void f(T, T);
4) template<typename T> void f(T*);
5) template<typename T> void f(T*, T);
6) template<typename T> void f(T, T*);
7) template<typename T> void f(int, T*);
8) template<> void f<int>(int);
9) void f(int, double);
10) void f(int);

- int i; double d;
- float ff; complex<double> c;
- **non-templates are always preferred over templates**

a) f(i); // invokes?
b) f<int>(i); // invokes?
c) f(i, i); // invokes?
d) f(c); // invokes?
e) f(i, ff); // invokes?
f) f(i, d); // invokes?
g) f(c, &c); // invokes?
h) f(i, &d); // invokes?
i) f(&d, d); // invokes?
j) f(&d); // invokes?
k) f(d, &i); // invokes?
l) f(&i, &i); // invokes?

Templates & function calls

1) template<typename T1, typename T2> void f(T1, T2);
2) template<typename T> void f(T);
3) template<typename T> void f(T, T);
4) template<typename T> void f(T*);
5) template<typename T> void f(T*, T);
6) template<typename T> void f(T, T*);
7) template<typename T> void f(int, T*);
8) template<> void f<int>(int);
9) void f(int, double);
10) void f(int);

- int i; double d;
- float ff; complex<double> c;

a) f(i); // 10
b) f<int>(i); // 8 – f<int> explicit
c) f(i, i); // 3
d) f(c); // 2
e) f(i, ff); // 1 why not 9? only exact match; ff is float not double.
f) f(i, d); // 9
g) f(c, &c); // 6
h) f(i, &d); // 7
i) f(&d, d); // 5
j) f(&d); // 4
k) f(d, &i); // 1
l) f(&i, &i); // 3

References

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