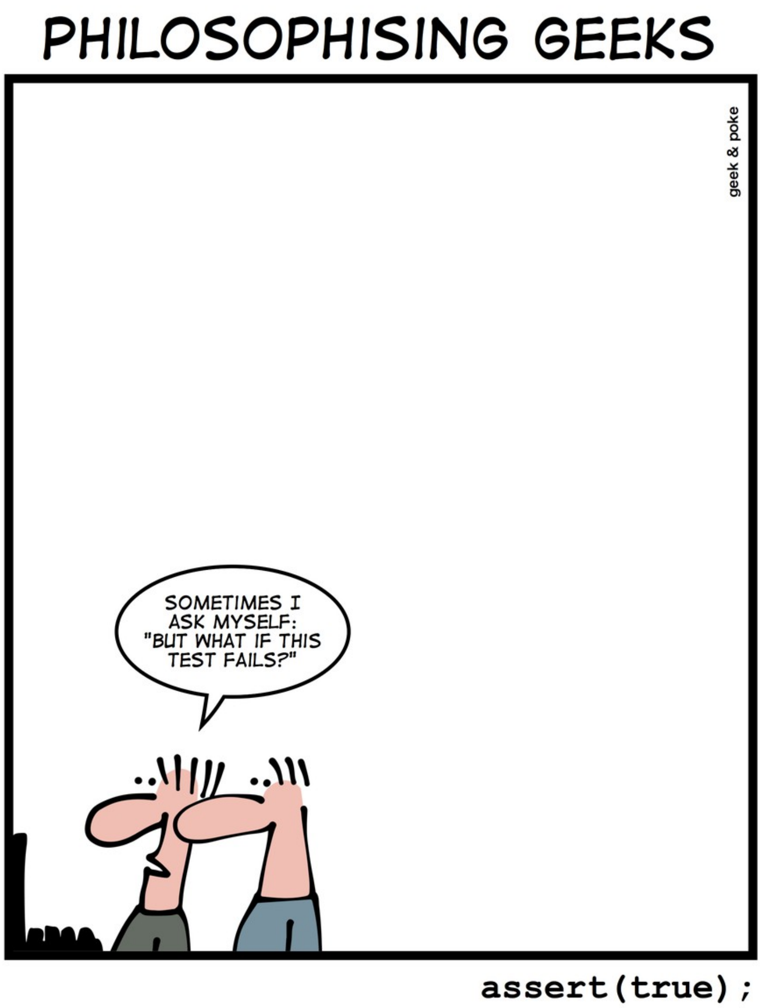
C++ Quiz



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? Why?
- T's type and why?
- T's type and why?
- T's type and why?

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?

  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&

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}”

Type deduction: decltype

- struct Point {

  int x, y;

  };
- Type of Point::x? - int
- Point p;

  const Point& cp = p';
- What is the type cp.x?
- Both
  - int
  - const int

- C++ solution

  decltype(cp.x) ~ int

  decltype((cp.x)) ~ const int&

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) ….

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

Templates & function calls

- **// What is the output of this program?**
- template <class T> void f(T &i) { std::cout << 1; }
- template <> void f(const int &i) { std::cout << 2; }
- int main() {
  - int i = 42;
  - f(i);
- }

Templates & function calls: Solution

- **// What is the open output of this program?**
- template <class T> void f(T &i) { std::cout << 1; }
- template <> void f(const int &i) { std::cout << 2; }
- int main() {
  - int i = 42;
  - f(i);
- }
- Solution: 1
- The templated function will be instantiated as void f(int&), which is a better match than f(const int&).

Templates & function calls II

- template <int i>

  void fun() {

  - i = 20;
  - std::cout << i;
- }
- int main() {
  - fun<10>();

  }

- Result:
- Compiler error in line "i = 20;"
- Non-type parameters must be const, so they cannot be modified.
- Compiler error: lvalue required as left operand of assignment i = 20;

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 & Inheritance

- struct Base{

      int x;
      template <int Trange>
      virtual void print()   {
          std::cout << Trange + x + 1; }
  };
- struct Derived : public Base {
  - template <int Trange>

        void print()   {
          std::cout << Trange + x + 2; }
- };
- int main() {

      Base* p = new Derived;
      p->x = 1;
      p->print<5>();
  }
- // Result -Error code doesn't compile!
- // Member function template cannot be virtual

- struct Base{

      int x;
      template <int Trange>
      void print()   {
          std::cout << Trange + x + 1; }
  };
- struct Derived : public Base {
  - template <int Trange>

        void print()   {
          std::cout << Trange + x + 2; }
- };
- int main() {

      Base* p = new Derived;
      p->x = 1;
      p->print<5>();
  }
- // Result – Base class - 7
- // For Derived* - Derived - 8

Template specialisation

- template <class T>
- T max (T &a, T &b) {

    std::cout << "generic";

    return (a > b)? a : b;
- }
- template <>
- int max <int> (int &a, int &b) {
  - std::cout << "specialised";

    return (a > b)? a : b;
- }
- int main () {

    int a = 10, b = 20;

    std::cout << max <int> (a, b);
- }

- Which one will be called?
- Result:
  - Specialised 20
- This is an example of template specialization. Sometime we want a different behaviour of a function/class template for a particular data type. For this, we can create a specialized version for that particular data type.

Namespace – Output?

- namespace foo {

    void bar() {

        x++;

    - std::cout << x;

  - }

    int x;

- }
- int main() {
  - foo:x = 0;
  - foo::bar();
- }

- namespace foo {
  - int x;
- }
- namespace foo {

    void bar() {

        x++;

        std::cout << x;

  - }
- }
- int main() {
  - foo:x = 0;
  - foo::bar();
- }

Namespace – Solution

- namespace foo {

    void bar() {

      x++;

    - std::cout << x;

  - }

    int x;

- }
- int main() {
  - foo:x = 0;
  - foo::bar();
- }
- // Error: Names used in a namespace must be declared before before their use

- namespace foo {
  - int x;
- }
- namespace foo {

    void bar() {

      x++;

      std::cout << x;

  - }
- }
- int main() {
  - foo:x = 0;
  - foo::bar();
- }
- // Result: 1

ADL (Argument Dependent Lookup)

- Which of the following functions are found when called in main during name lookup?
- namespace standards {

  struct datastructure { };

  void foo(const datastructure& ds) { }

  void bar() {}
- }
- int main() {

  standards::datastructure ds;

  foo(ds);

  bar();
- }

Koenig or ADL : Solutioin

- namespace standards {

  struct datastructure { };

  void foo(const datastructure& ds) { }

  void bar() {}
- }
- int main() {

  standards::datastructure ds;

  foo(ds);

  bar();
- }

- // Result = foo();
- This is called koenig lookup or argument dependent name lookup. In this case, the namespace 'standards' is searched for a function 'foo' because its argument 'ds' is defined in that namespace. For function 'bar', no additional namespaces are searched and the name is not found.

Lvalue reference - I

- int& fun() {

    static int x = 10;

    return x;

- }
- int main() {

    fun() = 30;

    std::cout << fun();

- }

- Guess the output

(a) 10

(b) 30

(c) Compiler error

Lvalue reference – I (Solution)

- int& fun() {

  static int x = 10;

  return x;
- }
- int main() {

  fun() = 30;

  std::cout << fun();
- }

Result : 30

- When a function returns by reference, it can be used as lvalue.
- Since x is a static variable, it is shared among function calls and the initialization line "static int x = 10;" is executed only once. The function call fun() = 30, modifies x to 30. The next call "cout << fun()" returns the modified value.

Lvalue reference - II

- int& fun() {

    int x = 10;

    return x;

- }
- int main() {

    fun() = 30;

    std::cout << fun();

- }

- Guess the output

(a) 10

(b) 30

(c) Compiler error

Lvalue reference – II (Solution)

- int& fun() {

    int x = 10;

    return x;
- }
- int main() {

    fun() = 30;

    std::cout << fun();
- }

Result : 10

- When a function returns by reference, it can be used as lvalue.
- Since x is a local variable, every call to fun() will have use memory for x and call “fun() = 30" will not effect on next call.
- Or
- **Segmentation fault**
- warning: reference to local variable ‘x’ returned [-Wreturn-local-addr]

Rvalue reference – Output?

- Case I
- int main() {

```
int x = int() = 3;
std::cout << x;
```

- }
- **Output:**
  - **Compiler Error**
  - **Undefined**
  - **3**

- Case II
- int main() {

```
int&& x = int();
x = 3;
std::cout << x;
```

- }
- **Output:**
  - **Compiler Error**
  - **Undefined**
  - **3**

Rvalue reference – Solution

- What is it output?
- int main() {

  int x = int() = 3;

  std::cout << x;
- }
- int() creates a temporary variable which is an rvalue. The temporary variable that is created can not be assigned to since it is an rvalue. Thus this code should not compile

- int main() {

  int&& x = int();

  x = 3;

  std::cout << x;
- }
- int() creates a temporary variable.
- int&& is a universal reference
- **Result: 3**

Exception handling

- class Base {};
- class Derived: public Base {};
- int main() {

  Derived d;

  try {

      throw d;

  }

  catch(Base b) {

      std::cout<<"Base Exception";

  }

  catch(Derived d) {

      std::cout<<"Derived Exception";

  }
- }

- Output
  - Compiler Error
  - Derived Exception
  - Base Exception

Exception handling - Solution

- class Base {};
- class Derived: public Base {};
- int main() {

    Derived d;

    try {

        throw d;

    }

    catch(Base b) {

        std::cout<<"Base Exception";

    }

    catch(Derived d) {

        std::cout<<"Derived Exception";

    }
- }

- Result
  - Base Exception
- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.
- If we put base class first then the derived class catch block will never be reached.
- In Java, catching a base class exception before derived is not allowed by the compiler itself.
- In C++, compiler might give warning about it, but compiles the code.

Exception handling - Scope

- void foo(int x) try {
  - int y = 2;
  - throw 1;
- }
- catch(int e) { }
- int main() {
  - foo(3);
- }

- Question: Which of the following variables can be accessed in foo's function try block handler?
  - x and y and e
  - x and e - correct
  - y and e
  - e

Exception scope - solution

- void foo(int x) try {
  - int y = 2;
  - throw 1;
- }
- catch(int e) { }
- int main() {
  - foo(3);
- }

- Result: x and e
- Function parameters are accessible in the try handler.
- Function local variables are NOT accessible in the try handler

Copy constructor

- struct A {
  - A(int x) : n{x} {}
  - int n;
- };
- int main() {
  - A a1;   // a
  - A a2(2);   // b
  - A a3(a2); // c
- }

- Which lines below should not compile?
  - Line a
  - Line b
  - Line c
  - Line a, b and c

Copy constructor

- struct A {
  - A(int x) : n{x} {}
  - int n;
- };
- int main() {
  - A a1;   // a
  - A a2(2);   // b
  - A a3(a2); // c
- }

- **Result:**
  - Line a
- If any user-declared constructor is present in the class, then no default constructor will be created implicitly.
- Additionally if no user declared copy constructor is declared, then an implicit copy constructor will be created by the compiler.
- In this example there is a user-declared constructor which prevents the default constructor from existing, but a copy constructor is still created implicitly.

Member initialization

- struct Foo {

  Foo(int n) : x{n++},
      y{n++}, z{n++} {}

  int x;

  int y;

  int z;
- };
- int main() {

  Foo f(3);

  std::cout << f.z;

  }

- Output?
  - 3
  - 4
  - 5
  - Undefined

Member initialization - Solution

- struct Foo {

  Foo(int n) : x{n++},
  y{n++}, z{n++} {}

  int x;

  int y;

  int z;
- };
- int main() {

  Foo f(3);

  std::cout << f.z;

  }

- Output
  - **5**
- There is a sequence point after the initalization of each base and member, thus the code is well-formed and defined.
  - x = 3
  - y = 4
  - z = 5

Pointer

- int main() {
  - char* ptr;
  - char str[] = "abcdefg";
  - ptr = str;
  - ptr += 5;
  - std::cout << ptr;
- }
- Output?

- Result: fg
- Pointer ptr points to string ‘fg’. So it prints fg.

Smart Pointers

- shared_ptr – share resources – std::shared_ptr<Base>
- unique_ptr – no sharing – std::unique_ptr<Base>
- // Raw pointer
- Main () {
  - …
  - Base* p = new Derived;
  - p→doSomething // if this thows exception → memory leak
  - delete p;
  - …
- }

Smart Pointers (cont...)

- If a memory is block is associated with shared_ptrs belonging to a different group, then there is an error.
- All shared_ptrs sharing the same reference count belong to a group.
-

- int main( ) {

  shared_ptr<int> sptr1(new int);

  shared_ptr<int> sptr2 = sptr1;

  shared_ptr<int> sptr3;

  sptr3 = sptr2;

  }

	Reference count
shared_ptr<int> sptr1(new int)	1
shared_ptr<int> sptr2 = sptr1	2
sptr3 = sptr2	3
When main ends -> sptr3 goes out of scope	2
When main ends -> sptr2 goes out of scope	1
When main ends -> sptr1 goes out of scope	0 -> The resource is released as the count goes down to zero.

Smart Pointers (cont...)

- int main( ) {
- int* p = new int;
- shared_ptr<int> sptr1(p);
- shared_ptr<int> sptr2(p);
- }

- This piece of code is going to cause an error because two shared_ptrs from different groups share a single resource.

	Reference count
shared_ptr<int> sptr1(p)	1
shared_ptr<int> sptr2(p)	1
main ends -> sptr1 goes out of scope	0 -> memory block pointed by sptr1 or p is destroyed as ref count is 0.
main ends -> sptr2 goes out of scope	0 -> **Crashhhhhhh!!!!!**. Because this tries to release memory block associated with sptr2, which is already being destroyed.

C++ Thread safety

- `some_obj' is a shared variable visible to multiple threads.
- // thread 1 (performs no additional synchronization)
- code_that_reads_from(some_obj); // passes some_obj by const &
- // thread 2 (performs no additional synchronization)
- code_that_modifies(some_obj); // passes some_obj by non-const &
- If threads 1 and 2 can run concurrently, is this code correctly synchronized if the type of some_obj is:
  - (a) int?
  - (b) string?
  - (c) vector<map<int,string>>?
  - (d) shared_ptr<widget>?
  - (e) mutex?
  - (f) condition_variable?
  - (g) atomic<unsigned>?

C++ Thread safety - Solution

- **(a) int? (b) string? (c) vector<map<int,string>>? (d) shared_ptr<widget>?**

  No. The code has one thread reading (via const operations) from some_obj, and a second thread writing to the same variable. If those threads can execute at the same time, that's a race and a direct non-stop ticket to undefined behavior land. The answer is to synchronize access to the variable, for example using a mutex:

- { // thread 1
-     lock_guard hold(mut_some_obj);   // acquire lock
-     code_that_reads_from(some_obj); // passes some_obj by const &
- }
- { // thread 2
-     lock_guard hold(mut_some_obj);   // acquire lock
-     code_that_modifies(some_obj);   // passes some_obj by non-const &
- }
- Virtually all types, including shared_ptr and vector and other types, are just as thread-safe as int; they're not special for concurrency purposes. It doesn't matter whether some_obj is an int, a string, a container, or a smart pointer… concurrent reads (const operations) are safe without synchronization, but the shared object is writeable, then the code that owns the object has to synchronize access to it.

C++ Thread safety - Solution

- **mutex? / condition_variable? / atomic<unsigned>?**
- Yes. For these types, the code is okay, because these types already perform full internal synchronization and so they are safe to access without external synchronization.
- In fact, these types had better be safe to use without external synchronization, because they're synchronization primitives you need to use as tools to synchronize other variables! And its turns out that that's no accident…
- *Guideline: A type should only be fully internally synchronized if and only if its purpose is to provide inter-thread communication (e.g., a message queue) or synchronization (e.g., a mutex).*

References

- Scott Meyers “The last thing D needs”
- Herb Sutter's Guru of the Week
- http://geeksquiz.com/c-plus-plus/
- http://www.mycppquiz.com/
- http://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers