Curso: C++ Moderno
Período: 1º semestre/2017
Aluno: Arthur Nunes de Paiva Santos Queiroz

Lista de exercícios Módulo 06: Dedução e inspeção de tipos com auto e decltype

Questões

Questão 1

Verifique, para os exemplos desses slides, se a dedução de tipos de sua IDE ou editor (utilize o hovering).

main.cpp

```cpp
#include <iostream>

int main()
{
   int i = 1;
   double d = 1.0


   std::cout   << "Type of (int + float): \t" << decltype(i+d) << std::endl;

   return 0;
}
```

Questão 2

Escreva um programa que, quando executado ou perante um erro de compilação, mostre o tipo exato (com qualificadores e referências) de determinada expressão.

Fonte: http://stackoverflow.com/questions/81870/is-it-possible-to-print-a-variables-type-in-standard-c
http://stackoverflow.com/questions/11310898/how-do-i-get-the-type-of-a-variable
http://geometrica.saclay.inria.fr/team/Marc.Glisse/tmp/printtype.cpp

main.cpp

```cpp
#include <iostream>
```

```cpp
#include "printtype.h"

struct Perso{};
typedef int (*fun)(double);
typedef Perso Tab[3][5][7];
typedef fun fun2(Tab*&);
typedef fun2** T2[9];
typedef Tab Perso::*U;

int main(){
   std::cout << Printtype<unsigned long*const*volatile*&()>().name() << std::endl;
   std::cout << Printtype<void(*&)(int(void),Perso&)>().name() << std::endl;
   std::cout << Printtype<const Perso&(...)>().name() << std::endl;
   std::cout << Printtype<long double(volatile int*const,...)>().name() << std::endl;
   std::cout << Printtype<int (Perso::**)(double)>().name() << std::endl;
   std::cout << Printtype<const volatile short Perso::*>().name() << std::endl;
   std::cout << Printtype<T2*(&)(long)>().name() << std::endl;
   std::cout << Printtype<U[][9]>().name() << std::endl;
}

// FIXME: const char[] is ambiguous!
```

Printtype.h

```cpp
// extern "C" not handled
#include <typeinfo>
#include <type_traits>
#include <string>
template<class...> struct Printtype;
template<> struct Printtype<> {
	std::string name() {
		return "";
	}
};
template<class T> struct Printtype<T> {
	std::string name(std::string append="") {
		return typeid(T).name()+append;
	}
};
template<class T1, class T2, class...U> struct Printtype<T1,T2,U...> {
	std::string name() {
		return Printtype<T1>().name()+","+Printtype<T2,U...>().name();
	}
};

#define BASIC_TYPE(X) \
template<> struct Printtype<X> { \
	std::string name(std::string append="") { \
		return std::string( #X )+append; \
	} \
}
BASIC_TYPE(void);
BASIC_TYPE(bool);
BASIC_TYPE(char);
BASIC_TYPE(signed char);
BASIC_TYPE(unsigned char);
BASIC_TYPE(short);
BASIC_TYPE(unsigned short);
```

```
BASIC_TYPE(int);
BASIC_TYPE(unsigned);
BASIC_TYPE(long);
BASIC_TYPE(unsigned long);
BASIC_TYPE(long long);
BASIC_TYPE(unsigned long long);
BASIC_TYPE(float);
BASIC_TYPE(double);
BASIC_TYPE(long double);
BASIC_TYPE(wchar_t);
BASIC_TYPE(char16_t);
BASIC_TYPE(char32_t);
#undef BASIC_TYPE

#define SUFFIX(X,Y) \
template<class T> struct Printtype<T X> { \
	std::string name(std::string append="") { \
		return Printtype<T>().name( Y +append); \
	} \
}
#define SUFFIX1(X) SUFFIX(X, #X )
#define SUFFIX2(X) SUFFIX(X, " " #X )

SUFFIX1(*);
SUFFIX1(&);
SUFFIX1(&&);
SUFFIX2(const);
SUFFIX2(volatile);
SUFFIX2(const volatile);
#undef SUFFIX
#undef SUFFIX1
#undef SUFFIX2

std::string eatspace(std::string s){
	if(!s.empty() && s[0]==' ') s.erase(0,1);
	return s;
}

template<class T,class...U> struct Printtype<T(U...)> {
	std::string name(std::string append="") {
		return Printtype<T>().name(
			(append.empty()?append:("("+eatspace(append)+")"))+
			"("+Printtype<U...>().name()+")");
	}
};

template<class T,class...U> struct Printtype<T(U...,...)> {
	std::string name(std::string append="") {
		std::string args=Printtype<U...>().name();
		if(!args.empty()) args+=",";
		args+="...";
		return Printtype<T>().name(
			(append.empty()?append:("("+eatspace(append)+")"))+
			"("+args+")");
	}
};

template<class T,class F> struct Printtype<T F::*> {
	std::string name(std::string append="") {
		return Printtype<T>().name(" "+
			Printtype<F>().name()+"::*"+append);
```

```
        }
};

namespace {
template<class U,bool=std::is_array<U>::value>
struct Print_from_array:Printtype<U>{};
template<class U> struct Print_from_array<U,true>{
        std::string name(std::string append="") {
                return Printtype<U>().name(append,true);
        }
};

template<class T>
struct Printarraytype {
        std::string name(std::string append,bool fromarray,std::string dim) {
                return Print_from_array<T>().name(
                        ((append.empty()||fromarray)?append:("("+eatspace(append)+")"))+
                        "[" + dim + "]");
        }
};
}

template<class T>
struct Printtype<T[]> {
        std::string name(std::string append="",bool fromarray=false) {
                return Printarraytype<T>().name(append,fromarray,"");
        }
};

template<class T,int d>
struct Printtype<T[d]> {
        std::string name(std::string append="",bool fromarray=false) {
                return Printarraytype<T>().name(append,fromarray,std::to_string(d));
        }
};
```

Questão 3

Tente descobrir os tipos deduzidos na função conv( ).

main.cpp

```
#include <iostream>
#include "../exe02/printtype.h"

void conv(){
        auto a = 1U;
        auto b = 9;
        auto c = 3.14;
        auto d = &b;
        auto& e = d;
        auto &&f = 10;
        auto g = a + b;
        auto h = a + c;
        auto i = a + d;
        const auto j = f;
        auto& k = e;
```

```
        auto& l = f;


   std::cout << "a => " << Printtype<decltype(a)>().name() << std::endl;
   std::cout << "b => " << Printtype<decltype(b)>().name() << std::endl;
   std::cout << "c => " << Printtype<decltype(c)>().name() << std::endl;
   std::cout << "d => " << Printtype<decltype(d)>().name() << std::endl;
   std::cout << "e => " << Printtype<decltype(e)>().name() << std::endl;
   std::cout << "f => " << Printtype<decltype(f)>().name() << std::endl;
   std::cout << "g => " << Printtype<decltype(g)>().name() << std::endl;
   std::cout << "h => " << Printtype<decltype(h)>().name() << std::endl;
   std::cout << "i => " << Printtype<decltype(i)>().name() << std::endl;
   std::cout << "j => " << Printtype<decltype(j)>().name() << std::endl;
   std::cout << "k => " << Printtype<decltype(k)>().name() << std::endl;
   std::cout << "l => " << Printtype<decltype(l)>().name() << std::endl;
}

int main()
{
        conv();

   return 0;
}
```

a => unsigned
b => int
c => double
d => int*
e => int*&
f => int&&
g => unsigned
h => double
i => int*
j => int const
k => int*&
l => int&

Questão 4

Explique o erro de compilação da função infer( ) e corrija-o sem remover o especificador auto.

Erros de compilação:

```
main.cpp:11:1: error: ‘infer’ function uses ‘auto’ type specifier without trailing return type
 )
 ^
main.cpp:11:1: note: deduced return type only available with -std=c++14 or -std=gnu++14
main.cpp: In function ‘auto infer(int, B*, D*)’:
main.cpp:15:9: error: inconsistent deduction for ‘auto’: ‘B*’ and then ‘D*’
  return d;
         ^
```

O primeiro/segundo erro origina-se do fato do c++11 não suportar inferência do tipo de retorno com o uso da keyword 'auto'.

O terceiro erro ocorre devido a haver ambiguidade em relação ao tipo de retorno da função, B* ou D*, a despeito de serem tipos

derivados.

Para corrigir os erros, especifica-se o tipo de retorno em função do primeiro argumento, utilizando decltype.

main.cpp

```
#include <iostream>
#include "../exe02/printtype.h"

struct B{};
struct D : B {};

auto infer(
        int n,
        B* b,
        D* d
) -> decltype(b)
{
        if(n % 2 == 0)
                return b;
        return d;
}

int main()
{
        B b;
        D d;

        infer(0, &b, &d);

   return 0;
}
```

Questão 5

Discuta as opções de qualificação de auto na função f( ) indicando casos de uso para cada uma delas.

main.cpp

```
void f(){
        std::vector<int> v {1, 2, 3};

        // No primeiro caso, os elementos são acessados por valor, sendo possível modificar a variável i sem alterar os objetos do vetor.
        for (auto i : v){}
```

```
        // A passagem por cópia permite modificar os elementos do vetor.
        for (auto& i : v){}

        // A passagem por referência constante permite ter acesso apenas leitura aos
elementos do vetor sem o custo de uma cópia.
        for (const auto& i : v){}

        // Na verdade, não vejo porque utilizar uma referência de rvalue neste caso.
        for (auto&& i : v){}
}
```

Questão 6

Implemente funções wrappers getValWrap( ) e getRefWrap( ) tal que seus especificadores de retorno sejam os mesmos, mas seus tipos retornados sejam os mesmos de getVa1( ) e getRef ( ).

main.cpp

```
#include <iostream>
#include "../exe02/printtype.h"

int global = 10;

int getVal() { return global; }
int& getRef() { return global; }

auto getValWrap() -> decltype(getVal())
{
        return getVal();
}

auto getRefWrap() -> decltype(getRef())
{
        return getRef();
}

int main(){

}
```