CS100
Introduction to Programming

Lecture 27: C++17, variadic templates, type deduction

Outline

- `std::optional`
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

Outline

- **`std::optional`**
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

std::optional

- Adds a flag to a type to achieve “nullable” types
- Use-case:
  - We need to express whether or not value has been set
- Example:

```
std::string UI::FindUserNick() {
  std::string result;
  if (nick_available)
    result = mStrNickName;
  return result;
}

…

std::string UserNick = UI->FindUserNick();
if (!UserNick.empty())
  show(UserNick);
```

`std::optional`

- Problem:
  - Sentinel values are used to differentiate nulltype
  - Examples:
    - **`int`**: -1
    - **`string`**: an empty string
    - pointer: **`nullptr`**
  - Sub-ideal
    - Prevents certain values to be used
  - Alternative:
    - **`std::unique_ptr<T>`**
    - Requires memory allocation on the heap

`std::optional`

- Solution:
  - Use **`std::optional`**
    - If the variable is available, it will return it
    - If not, it returns **`nullopt`**

```
std::optional<std::string> UI::FindUserNick() {
  if (nick_available)
    return { mStrNickName };
  return std::nullopt; // same as return { };
}

// use:
std::optional<std::string> UserNick = UI->FindUserNick();
if (UserNick)
  Show(*UserNick);
```

std::optional

- Creation of **`std::optional`**

```
// empty:
std::optional<int> oEmpty;
std::optional<float> oFloat = std::nullopt;

// direct:
std::optional<int> oInt(10);
std::optional oIntDeduced(10); // deduction guides

// make_optional
auto oDouble = std::make_optional(3.0);
auto oComplex = make_optional<std::complex<double>>(3.0, 4.0);
```

`std::optional`

- Creation of **`std::optional`**
  - Using option **`std::in_place`** saves creation of temporary!

```
// in_place
std::optional<std::complex<double>> o7{std::in_place, 3.0, 4.0};

// will call vector with direct init of {1, 2, 3}
std::optional<std::vector<int>> oVec(std::in_place, {1, 2, 3});

// copy/assign:
auto oIntCopy = oInt;
```

`std::optional`

- `std::in_place` works with custom types

```
struct Point {
  Point(int a, int b) : x(a), y(b) { }
  int x;
  int y;
};

std::optional<Point> opt{std::in_place, 0, 1};
// vs
std::optional<Point> opt{{0, 1}};
```

std::optional

- Returning **`std::optional`** from function
  - Return values are implicitly wrapped into **`std::optional`**!

```
std::optional<std::string> TryParse(Input input) {
  if (input.valid())
    return input.asString();
  return std::nullopt;
}
```

`std::optional`

- Accessing the stored value
  - **`operator*`** and **`operator->`**
    - similar to iterators. If there's no value the behavior is **undefined**!
  - **`value()`**
    - returns the value, or throws **`std::bad_optional_access`**
  - **`value_or(defaultVal)`**
    - returns the value if available, or **`defaultVal`** otherwise

- Checking if has value
  - Use **`has_value()`**
  - Check with **`if( optional )`**

std::optional

- Access examples

```
// by operator*
std::optional<int> oint = 10;
std::cout<< "oint " << *opt1 << '\n';

// by value()
std::optional<std::string> ostr("hello");
try {
  std::cout << "ostr " << ostr.value() << '\n';
} catch (const std::bad_optional_access& e) {
  std::cout << e.what() << "\n";
}

// by value_or()
std::optional<double> odouble; // empty
std::cout<< "odouble " << odouble.value_or(10.0) << '\n';
```

std::optional

- Most common way of usage

```
// compute string function:
std::optional<std::string> maybe_create_hello();
// ...

auto ostr = maybe_create_hello();
if (ostr)
  std::cout << "ostr " << *ostr << '\n';
else
  std::cout << "ostr is null\n";
```

std::optional

- Changing the value
  - Consider the class

```
#include <optional>
#include <iostream>
#include <string>

class UserName {
public:
  explicit UserName(const std::string& str) : mName(str) {
    std::cout << "UserName::UserName(\'";
    std::cout << mName << "\')\n";
  }
  ~UserName() {
    std::cout << "UserName::~UserName(\'";
    std::cout << mName << "\')\n";
  }
private:
  std::string mName;
};
```

std::optional

- Changing the values

```
int main() {
  std::optional<UserName> oEmpty;

  // emplace:
  oEmpty.emplace("Steve");

  // calls ~Steve and creates new Mark:
  oEmpty.emplace("Mark");

  // reset so it's empty again
  oEmpty.reset(); // calls ~Mark
  // same as:
  //oEmpty = std::nullopt;

  // assign a new value:
  oEmpty.emplace("Fred");
  oEmpty = UserName("Joe");
}
```

std::optional

- Use within comparisons

```
#include <optional>
#include <iostream>

int main() {
  std::optional<int> oEmpty;
  std::optional<int> oTwo(2);
  std::optional<int> oTen(10);
  std::cout << std::boolalpha;
  std::cout << (oTen > oTwo) << "\n";
  std::cout << (oTen < oTwo) << "\n";
  std::cout << (oEmpty < oTwo) << "\n";
  std::cout << (oEmpty == std::nullopt) << "\n";
  std::cout << (oTen == 10) << "\n";
}
```

std::optional

- A last example: Looping to find minimum
  - Old way

```
bool initialized = false;
float minimum;

for (auto v : values) {
  if (!initialized) {
    minimum = v;
    initialized = true;
  }
  else {
    if (v < minimum)
      minimum = v;
  }
}
```

std::optional

- A last example: Looping to find minimum
  - Another old way

```
float minimum = 1000000.0f;

for (auto v : values) {
  if (v< minimum)
    minimum = v;
}
```

std::optional

- A last example: Looping to find minimum
  - Using **std::optional**

```
std::optional<float> minimum;

for (auto v : values) {
  if (!minimum.has_value())
    minimum = v;
  else {
    if (v < minimum)
      minimum = v;
  }
}
```

Keyword `explicit`

- Going back to our example:

```
#include <optional>
#include <iostream>
#include <string>

class UserName {
public:
  explicit UserName(const std::string& str) : mName(str) {
    std::cout << "UserName::UserName(\'";
    std::cout << mName << "\')\n";
  }
  ~UserName() {
    std::cout << "UserName::~UserName(\'";
    std::cout << mName << "\')\n";
  }
private:
  std::string mName;
};
```

?

Keyword `explicit`

- Example:

```
class Foo {
 public:
  // single parameter constructor
  // can be used as an implicit conversion
  Foo (int foo) : m_foo (foo) {}

  int GetFoo () { return m_foo; }
 private:
  int m_foo;
};

void DoBar (Foo foo) {
  int i = foo.GetFoo ();
}

int main () {
  DoBar (42);
}
```

42 is not of type **`Foo`**, but there exists **`Foo`** constructor taking int → implicit conversion

Keyword `explicit`

- Example:

```
class Foo {
 public:
  // single parameter constructor
  // can no longer be used as an implicit conversion
  explicit Foo (int foo) : m_foo (foo) {}

  int GetFoo () { return m_foo; }
 private:
  int m_foo;
};

void DoBar (Foo foo) {
  int i = foo.GetFoo ();
}

int main () {
  DoBar (42);
}
```

Compiler error!

Outline

- std::optional
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

C++17: Structured bindings

- In a nutshell:
  - Define multiple variables in one go!

- Pre-C++17:
  - Use **`std::tuple`**

```
std::tuple<char, int, bool> mytuple() {
  char a = 'a';
  int i = 123;
  bool b = true;
  return std::make_tuple(a, i, b);
}
```

Structured bindings

- Assign to different variables:

```
char a;
int i;
bool b;

std::tie(a, i, b) = mytuple();
```

- With structured bindings!

```
auto [a, i, b] = mytuple();
```

Structured bindings

- Use with returned compound types (i.e. **`struct`**, **`pair`**, **`tuple`**)
- The old way:

```
std::map<char,int> mymap;
auto mapret = mymap.insert(std::pair('a', 100));
```

- The new way:

No need to look-up mapret.first & mapret.second

Template argument deduction

```
auto [itelem, success] = mymap.insert(std::pair('a', 100));
if (!success) {
  // Insert failure
}
```

Structured bindings

- Works together with range-based for-loops:
- The old way:

```
for (const auto & entry : mymap) {
  auto & key = entry.first;
  auto & value = entry.second;
  // Process entry
}
```

- The new “structured binding” way:

```
for (const auto & [key, value] : mymap) {
  // Process entry using key and value
}
```

Structured bindings

- Direct initialization with std::tuple
- The old way:

```
auto a = ‘a’;
auto i = 123;
auto b = true;
```

- The new way:

```
auto [a, i, b] = tuple(‘a’, 123, true);
// With no types needed for the tuple! -> See Slide 32
// No types needed at all!
```

Structured bindings

- Where else is direct initialization useful?
  - For-loops:
    - Initialization of multiple variables that belong to for-loop inside the for-loop statement!

- Example:

Definition outside for-loop undesirable

```
{
  istringstream iss(head);
  for (string name; getline(iss, name); ) {
    // Process name
  }
}
```

Structured bindings

- Limitation pre-C++17:

i and j need to be of same type!!

```
for (int i = 0, j = 100; i < 42; ++i, --j) {
  // Use i and j
}
```

- With structured bindings we can now do

```
for (
    auto [iss, name] = pair(istringstream(head), string {});
    getline(iss, name); ) {
  // Process name
}
```

Structured bindings

- Summary:
  - a single declaration that declares one or more local variables
  - that can have different types
  - whose types are always deduced using a single auto
  - assigned from a composite type

Outline

- std::optional
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

Pre-C++17

- Template Argument Deduction in a more general sense is the ability of templated functions or classes to determine the type of the passed arguments and from here derive the template types
- It has worked for functions since a while
  - Example:

```
template<class T>
void swap(T &x, T &y) {
  T t = std::move(x);
  x = std::move(y);
  y = std::move(t);
}
//…
char ch1, ch2;
swap(ch1,ch2);
```

Pre-C++17

- How to extend to classes?
- Example:

Need to specify template parameters, though it should not be necessary!

```
std::pair<int, double> p(2, 4.5);
```

- Alternative:

```
auto p = std::make_pair(2, 4.5);
```

- Confusing and artificial
- Inconsistent with non-template class construction
- Need for individual make_xyz functions for each case
- Have to implement your own make_xyz functions for custom types

C++17: Template argument deduction

- In C++17, we can now write

```
std::pair p1(2, 4.5);
auto p2 = std::pair(2, 4.5);
```

- Example:
  - Generating a tuple

```
auto mytuple() {
  char a = 'a';
  int i = 123;
  bool b = true;
  return std::tuple(a, i, b);  // No types needed
}
```

Template argument deduction

- Even better:
  - No types at all needed anymore!

```
auto mytuple() {
  return std::tuple('a', 123, true);
  // Auto type deduction from arguments
}
```

Outline

- std::optional
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

C++17: Selection initialization

- Allows for optional variable initializations inside **`if`** and **`switch`** statements
- Consider

```
for (int a = 0; a < 10; ++a) {
  // …
  // Good: a is limited to the scope of the for loop
}
```

- With **`if`** we do however have

```
{
  auto a = getval();
  if (a < 10) {
    // Use a
  }
  //a is still within this scope, though used only inside if
}
```

Selection initialization

- What is the problem?

```
int a;
if ((a = getval()) < 10) {
  // Use a
}
// …
// Much further on in the code
// a has the same value as previously

if (a == b) {
  //...
}
```

  - The intention was that a in the second `if` is a different variable, and an error has been committed by not initializing it
  - Will not be picked up by compiler!

Selection initialization

- In C++17, we can now write

```
if (auto a = getval(); a < 10) {
  //...
  //a is limited to scope of if body
}
```

- Follows same logic than with the for-loops
- Can be used with **`switch`** statements

```
switch (auto ch = getnext(); ch) {
  // case statements as needed
}
```

Selection initialization

- Back to our example

```
if (auto a = getval(); a < 10) {
  // Use a
}

// ...
// Much further on in the code - a is now not defined

if (a == b) {
  // ...
}
```

  - Compiler will throw error

Selection initialization

- Together with structured bindings, we now can do

```
if (
    auto [itelem, success] = mymap.insert(std::pair('a',100));
    success ) {
  // Insert success
}
```

- Or

```
if (auto [a, b] = myfunc(); a < b) {
  // Process using a and b
}
```

Outline

- std::optional
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

Variadic templates

- In C, we already had variadic functions
  - functions that take an arbitrary number of parameters
  - Example:
    - **`printf()`**
  - How does it work?

```c
// C program to demonstrate use of variable number of arguments
#include <stdarg.h>
#include <stdio.h>

// this function returns minimum of integer numbers passed.
// First argument is count of numbers.
int min(int arg_count, ...) {
  int i;
  int min, a;

  // va_list is a type to hold information about variable arguments
  va_list ap;

  // va_start must be called before accessing variable argument list
  va_start(ap, arg_count);

  // Now arguments can be accessed one by  using va_arg macro.
  // Initialize min as first argument in list
  min = va_arg(ap, int);

  // traverse rest of the arguments to find out minimum
  for (i = 2; i <= arg_count; i++)
    if ((a = va_arg(ap, int)) < min)
     min = a;

  // va_end should be executed before the function
  // returns whenever va_start has been previously used in that function
  va_end(ap);
  return min;
}
```

Variadic functions

- Example application:

```
// Driver code
int main() {
  int count = 5;
  printf("Minimum value is %d", min(count, 12, 67, 6, 7, 100));
  return 0;
}
```

```c
// C program to demonstrate working of
// variable arguments to find average
// of multiple numbers
#include <stdarg.h>
#include <stdio.h>

int average(int num, ...) {
  va_list valist;
  int sum = 0, i;

  va_start(valist, num);
  for (i = 0; i < num; i++)
    sum += va_arg(valist, int);

  va_end(valist);
  return sum / num;
}

// Driver code
int main() {
  printf("Average of {2, 3, 4} = %d\n",
                       average(2, 3, 4));
  printf("Average of {3, 5, 10, 15} = %d\n",
                    average(3, 5, 10, 15));
  return 0;
}
```

Variadic functions

- Drawbacks:
  - Number and type of arguments must be known upfront
  - How does **`printf`** work?

    - **`printf(char* str,…)`**
    - str is a string with placeholders for variables indicating their type
      - %s
      - %f
      - %d
    - Parsing this string will tell the number/type of the expected arguments

Variadic templates

- Example:
  - **`std::tuple<int,float,string>`**
  - How is it done?

- Before variadic templates:
  - Emulations
    - Many defaulted template parameters
    - Sets of overloaded function templates, each one taking a different number of template parameters
  - Code duplication is avoided by preprocessor metaprogramming

Variadic templates

- Before variadic templates
  - Example of extra defaulted template parameters

```
struct unused;
template<typename T1 = unused, typename T2 = unused,
         typename T3 = unused, typename T4 = unused,
         /* up to */ typename TN = unused> class tuple;
```

- Presuming N is large enough, we can define

```
typedef tuple<char, short, int, long, long long> integral_types;
```

- Drawbacks:
  - would leave N-5 unused parameters
  - Need to be somehow ignored by implementation

Variadic templates

- Extra defaulted template parameters would need
  - (Clear limitation in terms of maximum number of parameters)

```
template<>
class tuple<> {
  /* handle zero-argument version. */
};

template<typename T1>
class tuple<T1> {
  /* handle one-argument version. */
};

template<typename T1, typename T2>
class tuple<T1,T2> {
  /* handle two-argument version. */
};
```

Variadic templates

- Variadic templates let us explicitly state that class accepts variable number of template parameters

```
template<typename... Elements> class tuple;
```

- ... indicates that Elements is ***template parameter pack***
  - Multiple arguments are packed into a single parameter pack
  - Can be used with non-type template parameters as well

```
template<typename T, unsigned PrimaryDimension, unsigned... Dimensions>
class array { /* implementation */ };
array<double, 3, 3> rotation matrix; // 3x3 rotation matrix
```

Variadic templates

- To use the template parameter packs, we must *unpack* them
  - They are subsequently passed to another template
  - The idea of unpacking arbitrary packs involves recursive, one-by-one unpacking

- Example:
  - Template that simply counts the number of arguments it is given

```
template<typename... Args> struct count;
```

Variadic templates

```
template<typename... Args> struct count;
```

- Step 1: Define basis case specialization
  - If count has 0 zero arguments, internal value is set to 0

```
template<>
struct count<> {
  static const int value = 0;
};
```

Variadic templates

```
template<typename... Args> struct count;
```

- Step 2: Define recursive case
  - Takes 1 plus N-1 arguments, peels one off, and initializes specialization with remaining **unpacked** arguments

Unpacking of parameter pack into separate arguments

```
template<typename T, typename... Args>
struct count<T, Args...> {
  static const int value = 1 + count<Args...>::value;
};
```

Variadic templates

- Recursive algorithm! Partial specialization …
  - … pulls off one template parameter
  - … places remaining arguments into Args
  - … defines another partial specialization with only Args
- When Args is empty, instantiation selects full specialization **`count<>`** → recursion is terminated
- **`count<char,short,int>`** would select partial specialization
  - Binding char to T
  - Binding short and int as a parameter pack to Args

Variadic templates

- Tuple with recursive inheritance

```
template<typename… Elements> class tuple;
template<typename Head, typename… Tail>
class tuple<Head, Tail…> : private tuple<Tail…> {
  Head head;
 public:
  /* implementation */
};

template<> class tuple<> { /* zero-tuple implementation */ };
```

Variadic templates

- Allow arbitrary number of arguments to be passed to a function

Function parameter pack

```
template<typename... Args> void eat(Args... args);
```

- Now we can implement a C++-style, type-safe printf function

Variadic templates

```
void printf(const char* s) {
  while (*s) {
    if (*s == ′%′ && *++s != ′%′)
      throw std::runtime error(″invalid format string: missing args″);
    std::cout << *s++;
  }
}

template<typename T, typename... Args>
void printf(const char* s, T value, Args... args) {
  while (*s) {
    if (*s == ′%′ && *++s != ′%′) {
      std::cout << value;
      return printf(++s, args...);
    }
    std::cout << *s++;
  }
  throw std::runtime error(″extra arguments provided to printf″);
}
```

Outline

- std::optional
- C++17: Structured Bindings
- C++17: Template Argument Deduction
- C++17: Selection Initialization
- Variadic templates
- Understanding type deduction

Understand Type Deduction

- In C++ 98, type deduction used only for templates
  - Generally just works
  - Detailed understanding rarely needed
- In C++ 11, scope expands:
  - Auto variables, universal references, lambda captures and returns, **`decltype`**
  - Just works less frequently
- In C++ 14, scope expands further:
  - Function return types, lambda init captures
  - Same rulesets, but more usage contexts (and chances for confusion)

(`auto-related`) Template Type Deduction

- General Problem:

```
template<typename T>
void f(ParamType param);
f(expr);                          //deduce T and ParamType from expr
```

- Given type or expr, what are these types?
  - **`T`**:
    - The deduced type
  - **`ParamType`**:
    - Often different from T (e.g. **`const T&`**)
- Three general cases:
  - **`ParamType`** is a reference or pointer, but not a universal reference
  - **`ParamType`** is a universal reference
  - **`ParamType`** is neither reference nor pointer

Non-URef Reference/Pointer Parameters

- Type deduction very simple:
  - If **`expr's`** type is a reference, ignore that
  - Pattern-match **`expr's`** type against **`ParamType`** to determine **`T`**

```
template<typename T>
void f(T& param);                  //param is a reference
int x = 22;                        //int
const int cx = x;                  //copy of int
const int &rx = x;                 //ref to const view of int
f(x);          //T ≡ int, param's type = int&
f(cx);         //T ≡ const int, param's type ≡ const int&
f(rx);         //T ≡ const int, param's type ≡ const int&
```

- Note : **`T`** not a reference

Non-URef Reference/Pointer Parameters

- `ParamType` of `const T&` ⇒ `T` changes, but param’s type doesn’t:

```
template<typename T>
void f(const T& param);
int x = 22;                       //as before
const int cx = x;                 //as before
const int& rx = x;                //as before
f(x);          //T ≡ int, param's type = const int&
f(cx);         //T ≡ int, param's type ≡ const int&
f(rx);         //T ≡ int, param's type ≡ const int&
```

- Note : `T` not a reference

Non-URef Reference/Pointer Parameters

- Behavior with pointers essentially the same:

```
template<typename T>
void f(T* param);                    //param now a pointer
int x = 22;                          //int
const int *pcx = &x;                 //ptr to const view of int
f(&x);          //T ≡ int, param's type ≡ int*
f(pcx);         //T ≡ const int, param's type ≡ const int*
```

- Note : `T` not a pointer
- Behavior of `const T*` parameters as you'd expected

`auto` and Non-URref Reference/Pointer Variables

- `auto` plays role of `T`:

```
int x = 22;                      //as before
const int cx = x;                //as before
const int& rx = x;               //as before
auto& v1 =x;                   //v1's type ≡ int& (auto ≡ int)
auto& v2 =cx;                  //v2's type ≡ const int&
                               //(auto ≡ const int)
auto& v3 =rx;                  //v3's type ≡ const int&
                               //(auto ≡ const int)
const auto& v4 = x;            //v4's type ≡ const int&
                               //(auto ≡ int)
const auto& v5 = cx;           //v5's type ≡ const int&
                               //(auto ≡ int)
const auto& v6 = rx;           //v6's type ≡ const int&
                               //(auto ≡ int)
```

What are Universal References?

- Consider

```
void foo(int&& i);

foo(1);        // OK, 1 is an rvalue

int k = 0;
foo(k);        // error: cannot bind 'int' lvalue to 'int&&'
```

What are Universal References?

- However, the following works

Binds to both rvalues and lvalues

```
template <typename T>
void bar(T&& i);

bar(1);        // OK

int k = 0;
bar(k);        // OK (?!)
```

- Called: universal reference!
- Alternative name: forwarding reference

Universal References

```
template<typename T>
void f(T&& param);
f(expr);
```

- Treated like “normal” reference parameters, except:
  - If **`expr`** is lvalue with deduced type **`E`**, **`T`** deduced as **`E&`**
    - Reference-collapsing yields type **`E&`** for param

```
int x = 22;                //as before
const int cx = x;          //as before
const int& rx = x;         //as before
f(x);               // x is lvalue ⇒ T = int&,
                    // param's type ≡ int&
f(cx);              // cx is lvalue ⇒ T = const int&
                    // param's type ≡ const int&
f(rx);              // rx is lvalue ⇒ T = const int&
                    // param's type =const int&
f(22);              // 22 is rvalue ⇒ no special handling;
                    // T ≡ int, param's type is int&&
```

By-Value Parameters

- Deduction rules a bit different (vis-à-vis by reference/by-pointer):
  - As before, if **`expr's`** type is a reference, ignore that
  - If **`expr`** is const or volatile, ignore that
  - **`T`** is the result

```
template<typename T>
void f(T param);               //param passed by value
int x = 22;                    //as before
const int cx = x;              //as before
const int& rx = x;             //as before
f(x);                          // T = int, param's type = int
f(cx);                         // T = int, param's type = int
f(rx);                         // T = int, param's type = int
```

- **`expr's`** reference-/const-qualifiers always dropped in deducing **`T`**.

Non-Reference Non-Pointer `autos`

- `auto` again plays role of T:

```
int x = 22;                  //as before
const int cx = x;            //as before
const int& rx = x;           //as before
auto v1 = x;                 //v1's type ≡ int (auto ≡ int)
auto v2 = cx;                //v2's type ≡ int (auto ≡ int)
auto v3 = rx;                //v3's type ≡ int (auto ≡ int)
```

- Again, `expr's` reference-/const-qualifiers always dropped in deducing **T**
  - auto never deduced to be a reference.
    - It must be manually added.
      - If present, use by-reference rulesets

```
auto v4 = rx;          //v4's type ≡ int
auto& v5 = rx;         //v5's type ≡ const int&
auto&& v6 = rx;        //v6's type ≡ const int&
                       //(rx is lvalue)
```

`decltype`

- With **`decltype`**, we can get the type of an existing named variable so we can define a new variable of same type:

```
size_t i=20000000;
decltype(i) j{9};
```

`decltype` Type Deduction

- **`decltype(name)`** = declared type of name. Unlike **`auto`**;
- Never strips **`const`**/**`volatile`**/references

```
int x = 10;                  // decltype(x) ≡ int
const auto& rx = x;          // decltype(rx) ≡ const int&
```

Function Return Type Deduction

- In C++11:
  - Limited: single-statement lambdas only
- In C++14:
  - Extensive: all lambdas + all functions
    - Understanding type deduction more important than ever
- Deduced return type specifiers
  - **`auto`**: Use template (not auto!) type deduction rules
  - **`decltype(auto)`** : Use **`decltype`** type deduction rules

Function Return Type Deduction

- Sometimes auto is correct:

```
auto lookupValue(context information) {
    static std::vector<int> values = initValues();
    int idx = …; //compute index into values from context info
    return values[idx];
}
```

- Returns **`int`**
- **`decltype(auto)`** would return **`int&`**
  - Would permit caller to modify values!

```
lookupValue(myContextInfo) = 0;    // shouldn't compile!
```

Function Return Type Deduction

- Sometimes **`decltype(auto)`** is correct:

```
decltype(auto) authorizeAndIndex(std::vector<int>& v, int idx) {
  authorizeUser();
  return v[idx];
}
```

- Returns **`int&`**
- **`auto`** would return **`int`**
  - Wouldn't permit caller to modify **`std::vector`**

```
authorizeAndIndex(myVec, 10) = 0;     // should compile!
```

Function Return Type Deduction

- **`decltype(auto)`** sensitive to function implementation:

```
decltype(auto) lookupValue(  context information ) {
    static std::vector<int> values = initValues();
    int idx = …; //compute index into value from context info
    auto retVal = values[idx];    // retVal’s type is int
    return retVal;                // returns int
}

decltype(auto) lookupValue(  context information ) {
    static std::vector<int> values = initValues();
    int idx = …; //compute index into value from context info;
    auto retVal = values[idx];      //retVal’s type is int
    return (retVal);        // returns int& (to local variable)
}
```

Function Return Type Deduction

- Rules of thumb:
  - Use `auto` if a reference type would never be correct
  - Use `decltype(auto)` only if a reference type could be correct
    - “Perfect returning”