COMP6771
Advanced C++ Programming

Week 6
Part One: Function Templates

2016

www.cse.unsw.edu.au/˜cs6771

Constants

Two notions of immutability:

- const: A promise not to change this value.
  - Used primarily to specify interfaces, so that data can be passed to functions without the fear of it being modified.
  - The compiler enforces the promise made.
- constexpr: An indication for it to be evaluated at compile time.
  - Used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted and for performance

constexpr and Constant Expressions

- constexpr specifies that the value of a variable or function can be computed at compile time
- A constant expression is an expression whose value cannot change and that can be evaluated at compile-time, including:
  - A literal
  - A const object that is initialised from a constant expression

```
const int max_files = 20;          // constant expr
const int limit = max_files + 1;  // constant expr
int staff_size = 27;              // No as it can be changed
const int sz = get_size();        // No as shown below
```

```
x.cpp:
#include<iostream>

int getsize() {
  int i;
  std::cin >> i;
  return i;
}

extern const int z = getsize();
```

```
y.cpp:
#include<iostream>

extern const int z;

int main() {
  std::cout << z << std::endl;
}
```

constexpr

- Verified by the compiler to be a constant expression
- Examples:

```
constexpr int max_files = 20;          // YES
constexpr int limit = max_files + 1; // YES

constexpr int multiply (int x, int y) { return x * y; }
const int val = multiply(10, 10); // YES

constexpr int sz = size();
   // YES only if size is a constexpr function.
```

- Pointers and `constexpr`

```
const int *p = nullptr; // p is a pointer to a const int
                        // not a constant expression
constexpr int *q = nullptr; // q is a const pointer to int
// SAME AS
int * const q = nullptr;    // YES
```

`constexpr` relevant to top-level constness only

constexpr Functions

- The functions that can be used as constant expressions
- Restrictions: http://en.cppreference.com/w/cpp/language/constexpr
  - The return and parameter types must be literal types
  - The function body must contain only:
    - type/aliasing declarations,
    - null statements
    - a single return statement that evaluates to a constant expression during the function invocation substitution.
    - ...

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz(); // ok: foo is a constant expr
```

- `constexpr constructors` as a special case (§7.5.6)
- Literal types: the types allowed in a `constexpr`, including:
  - Arithmetic, reference and pointer types
  - literal classes (§7.5.6)

Literal Classes

- Aggregate classes:
  - all its members are public
  - no constructors
  - no in-class initialisers

```
struct Dara {
  int ival;
  string s;
};

Data d = { 0, "Anna" };
```

- An aggregate class is a literal class if all its members are literal types
- A non-aggregate class can also be a literal class if it satisfies the requirements listed in Page 299 (§7.5.6)

Static vs. Dynamic Polymorphism

- Static (compile-time) polymorphism:
  - Function overloading
  - Templates: polymophical across unrelated types
    - Widely used in libraries (e.g., the STL library) to achieve generality, flexibility and efficiency
    - STL containers, iterators and algorithms are templates

```
std::vector<int> vi;
std::vector<float> vf;
std::sort(vi.begin(), vi.end())
std::sort(vf.begin(), vf.end())
```

- Dynamic (runtime) polymorphism: virtual functions are polymophic across types related by inheritance

What Is Generic Programming (GP)?

- GP is about generalising software components so that they are independent of any particular type
- Function and class templates are the foundation of GP
- STL is an example of generic programming
- Week 6
  - Function templates
  - Class templates

Assignment 3 is on templates and iterators.

Part I: Function Templates

- Template parameter list: type and nontype parameters
- Template argument deduction
- Explicit template arguments
- Specialisation
- Overloading function templates
- Function template instantiation: point of instantiation
- Name resolution

Why Function Templates?

- As a strongly-typed language, C++ requires:

```
int min(int a, int b) { 1
  return a < b ? a : b;
}
double min(double a, double b) { // 2
  return a < b ? a : b;
}
... more for other types ...
```

- Call resolution due to function overloading:

```
min(1, 2); // 1
min(1.1, 2.2); // 2
...
```

- C++ FAQ-lite 9.5 (macros are evil)
- Read: http://www.gotw.ca/gotw/077.htm

What Are Function Templates?

- Definition:

```
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

- Uses:

```
min(1, 2)     // int min(int, int)
min(1.1, 2.2) // double min(double, double)
...
```

NB

A function template is a prescription for the compiler to generate particular instances of a function varying by type

Template Parameter List

```
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

type-dependent interface

type-independent body

- Separation of type-dependent from type-independent parts
- T:
  - called a template type parameter
  - a placeholder for any built-in or user-defined type
  - Historically, class can also be used instead of typename

Type and Nontype Parameters

```
#include<iostream>

template <typename T, int size>
T findmin(const T (&a)[size]) {
  T min = a[0];
  for (int i = 1; i < size; i++)
    if (a[i] < min) min = a[i];
  return min;
}

int main() {
  int x[] = { 3, 1, 2 };
  double y[] = { 3.3, 1.1, 2.2, 4.4};
  std::cout << "min of x = " << findmin(x) << std::endl;
  std::cout << "min of y = " << findmin(y) << std::endl;
}
```

- T: a type parameter (an unknown type)
- size: a nontype parameter (an unknown value)
- The compiler deduces T and size from x for a

Type and Nontype Parameters

- The compiler generates two instances of min:

```
int findmin(const int (&a)[3]) {
  int min = a[0];
  for (int i = 1; i < 3; i++)
    if (a[i] < min) min = a[i];
  return min;
}

double findmin(const double (&a)[4]) {
  double min = a[0];
  for (int i = 1; i < 4; i++)
    if (a[i] < min) min = a[i];
  return min;
}
```

Problem: code explosion — different instances generated even for arrays of ints with different sizes

Type Equivalence and Nontype Parameter

```
template <typename T, int size>
T findmin(const T (&a)[size]) {
  T min = a[0];
  for (int i = 1; i < size; i++)
    if (a[i] < min) min = a[i];
  return min;
}

int main() {
  int x[] = { 3, 1, 2 };
  const int sz = 3;
  int y[sz];     // okay because this is a constant expression
  findmin(x);    // instantiates findmin(const int (&)[3])
  findmin(y);    // same instantiation
}
```

Expressions that evaluate to the same value are considered equivalent template arguments for nontype parameters

Inclusion Compilation Model

```
// user-file1.cpp:
#include "min.h"
min(1, 2);
min(1.1, 2.2);
```

```
// user-file2.cpp:
#include "min.h"
min(1, 2);
min(p, q);
```

```
// min.h:
// other declarations
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

- Templates in header files and compile only .cpp files
- Different instantiations may be generated but the compiler should behave as if only one int min(int, int) were instantiated
- Drawbacks:
  - implementation details available in .h files
  - compiling the same template many times can be slow

Function Template Instantiation

```
template <typename T>
   T min(T a, T b) {
     return a < b ? a : b;
}
std::cout << min(1, 2) << std::endl;  1
double (*pf)(double, double) = &min;  2
std::cout << pf(1.1, 2.2) << std::endl;
```

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

```
double min(double a, double b) {
  return a < b ? a : b;
}
```

- Instantiated when invoked or when its address taken
- 1 and 2: points of instantiation

Explicit Instantiations (Not Recommended)

```cpp
// min.h: declaration of min, should also define it in this file
template <typename T> T Min(T a, T b);
```

```cpp
// min-instances.cpp
#include "min.h"

// defintion of min is here.
template <typename T>
T Min(T a, T b) {
  return a < b ? a : b;
}

// explicit instantiations - without these this code will not link.
template int Min<int>(int, int);
template double Min<double>(double, double);
```

```cpp
// min-user.cpp
#include <iostream>
#include "min.h"

int main() {
   std::cout << Min(1, 2) << std::endl;
   std::cout << Min(1.1, 2.2) << std::endl;
}
```

- The compiler doesn't instantiate Min since it does not see any template definition when compiling min-user.cpp
- The linker will eventually create an executable code

`extern` and Explicit Instantiations

- Explicit instantations:

```
file1.cpp:

#include "min.h"
...
template int min(int, int); // <-- Instantiate it here
...
```

- Use one instantiated in another file:

```
file2.cpp:

#include "min.h"
Use this instantiation generated somewhere else
extern template int min(int, int);
...
min(1, 2); // Will not produce an instantiation in this file
...
```