

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

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

```
1 x.cpp:
2 #include<iostream>
3
4 int getsize() {
5     int i;
6     std::cin >> i;
7     return i;
8 }
9
10 extern const int z = getsize();
```

```
1 y.cpp:
2 #include<iostream>
3
4 extern const int z;
5
6 int main() {
7     std::cout << z << std::endl;
8 }
```

constexpr

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

```
1 constexpr int max_files = 20;           // YES
2 constexpr int limit = max_files + 1;    // YES
3
4 constexpr int multiply (int x, int y) { return x * y; }
5 const int val = multiply(10, 10); // YES
6
7 constexpr int sz = size();
8 // YES only if size is a constexpr function.
```

- Pointers and constexpr

```
1 const int *p = nullptr; // p is a pointer to a const int
2                          // not a constant expression
3 constexpr int *q = nullptr; // q is a const pointer to int
4 // SAME AS
5 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.
 - ...

```
1 constexpr int new_sz() { return 42; }  
2 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

```
1 struct Data {  
2     int ival;  
3     string s;  
4 };  
5  
6 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**: polymorphical 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 polymorphic 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:

```
1 int min(int a, int b) { 1  
2     return a < b ? a : b;  
3 }  
4 double min(double a, double b) { // 2  
5     return a < b ? a : b;  
6 }  
7 ... 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;
}
```

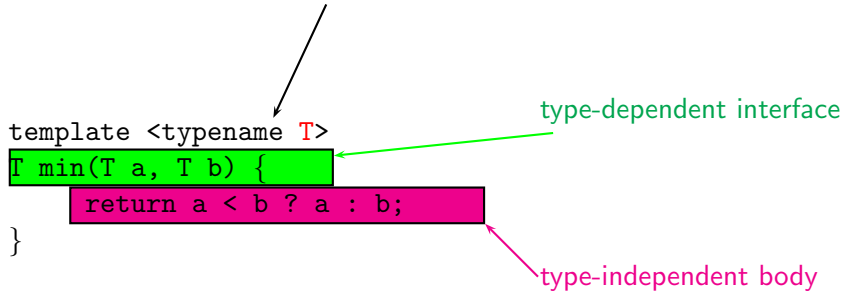
- Uses:

```
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>
```

The diagram shows a C++ template function definition. The text is as follows: `template <typename T>` (with `T` in red), `T min(T a, T b) {` (highlighted in green), `return a < b ? a : b;` (highlighted in pink), and `}`. Annotations include: a black arrow pointing to `T` in the template parameter list; a green arrow pointing to the green highlight with the text "type-dependent interface"; and a pink arrow pointing to the pink highlight with the text "type-independent body".

type-dependent interface

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

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

```
1  #include<iostream>
2
3  template <typename T, int size>
4  T findmin(const T (&a)[size]) {
5      T min = a[0];
6      for (int i = 1; i < size; i++)
7          if (a[i] < min) min = a[i];
8      return min;
9  }
10
11 int main() {
12     int x[] = { 3, 1, 2 };
13     double y[] = { 3.3, 1.1, 2.2, 4.4};
14     std::cout << "min of x = " << findmin(x) << std::endl;
15     std::cout << "min of y = " << findmin(y) << std::endl;
16 }
```

- 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**:

```
1  int findmin(const int (&a)[3]) {  
2      int min = a[0];  
3      for (int i = 1; i < 3; i++)  
4          if (a[i] < min) min = a[i];  
5      return min;  
6  }  
7  
8  double findmin(const double (&a)[4]) {  
9      double min = a[0];  
10     for (int i = 1; i < 4; i++)  
11         if (a[i] < min) min = a[i];  
12     return min;  
13 }
```

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

Type Equivalence and Nontype Parameter

```
1  template <typename T, int size>
2  T findmin(const T (&a)[size]) {
3      T min = a[0];
4      for (int i = 1; i < size; i++)
5          if (a[i] < min) min = a[i];
6      return min;
7  }
8
9  int main() {
10     int x[] = { 3, 1, 2 };
11     const int sz = 3;
12     int y[sz];    // okay because this is a constant expression
13     findmin(x);   // instantiates findmin(const int (&)[3])
14     findmin(y);   // same instantiation
15 }
```

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);
```

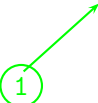
```
// min.h:  
// other declarations  
template <typename T>  
T min(T a, T b) {  
    return a < b ? a : b;  
}
```

- 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;  
    }
```

```
int min(int a, int 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;
```



```
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)

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

```
1 // min-instances.cpp
2 #include "min.h"
3
4 // definition of min is here.
5 template <typename T>
6 T Min(T a, T b) {
7     return a < b ? a : b;
8 }
9
10 // explicit instantiations - without these this code will not link.
11 template int Min<int>(int, int);
12 template double Min<double>(double, double);
```

```
1 // min-user.cpp
2 #include <iostream>
3 #include "min.h"
4
5 int main() {
6     std::cout << Min(1, 2) << std::endl;
7     std::cout << Min(1.1, 2.2) << std::endl;
8 }
```

- 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 instantiations:

```
1 file1.cpp:
2
3 #include "min.h"
4 ...
5 template int min(int, int); // <-- Instantiate it here
6 ...
```

- Use one instantiated in another file:

```
1 file2.cpp:
2
3 #include "min.h"
4 Use this instantiation generated somewhere else
5 extern template int min(int, int);
6 ...
7 min(1, 2); // Will not produce an instantiation in this file
8 ...
```