CSCI251/CSCI851 Spring-2021
Advanced Programming (**S6a**)

Generic Programming I:
Function templates and compile time functionality

Outline

- An introduction to generic programming.
- Function templates and template functions.
- Creating function templates.
- Calling function templates.
- Multiple arguments.
- Overloading function templates.
- Multiple generic types in function templates.
- Explicit calling.
- Default template arguments.
- Nontype template parameters.
- Compile time computation.

An introduction to generic programming

- In generic programming constructs are written in terms of *to-be-specified-later* types that are then *instantiated* when needed for specific types provided as parameters.
- We can have instantiations associated with multiple different types, but we just have the one abstract representation of the construct, a blueprint.
- We will look at functions first, then classes.

Function templates and template functions

- These are not the same and it’s easy to get them mixed up.
- The function templates are the prototypes expressed in terms of the *to-be-specified-later* types.
- The template functions are instantiations of those templates.

https://docs.microsoft.com/en-us/cpp/cpp/templates-cpp?view=msvc-160#:~:text=Default%20template%20arguments.%20Class%20and%20function%20templates%20can,T%2C%20class%20Allocator%20%3D%20allocator%3CT%3E%3E%20class%20vector%3B%20

The task …

- Task: Create functions to take the negative of any type.
- **Solution 1**:
  - A pre-processor directive to define MACRO:

```
#define reverse(x)  (-(x))
```

  - Problem:
    - Lack of type checking.
- **Solution 2**:
  - Function overloading:
    - Create functions for all needed types of x.

```
float reverse(float x) {
        return -x;
}
```

```
int reverse(int x) {
   return -x;
}
```

```
double reverse(double x) {
        return -x;
}
```

- Listing all possible types of the variable is tedious and sometime unreasonable for functions which are supposed to work on a wide range of argument types.
- But the overloaded functions clearly follow a particular “pattern”, they implement the same algorithm.

```
variableType reverse(variableType x)
{
      return -x;
}
```

- Ideally, you would like to define just one function with a variable accepting any data type.
- And we can do this, with a function template!

Creating function templates

```
template <typename T>
T reverse( T x ) {
      return -x;
}
```

```
template <class T>
T reverse( T x ) {
      return -x;
}
```

- T is a template parameter, this is a to-be-specified-later type, and we can generally specify a list of such arguments.
- The use of `typename` is standard now in the specification of this template parameter list.

Calling template functions

- When we call a function template, the compiler determines the type of the actual argument passed.

```
double amount = -9.86;
amount = reverse(amount);
```

- The designation of the parameterized type is implicit: determined by the compiler's ability to determine the argument type.
- The compiler generates code for different functions as it needs, depending on the function calls.
  - This is not in the preprocessor so you won't see it with the `-E` flag.

```
double reverse(double x) {
        return -x;
}
```

- The **function template** is a function blueprint that uses generic data types.
  - It defines a group of functions which may be generated by the compiler with different types of function parameters.
- You write a function template and the compiler generates one or more **template functions**, assuming there is at least one call to the function template.

```cpp
#include <iostream>
using namespace std;

template <typename T>
T reverse (T x)
{
  return (-x);
}

int main() {
  int a = 10;
  float b = 15.4;

  cout<<reverse(a)<<endl;
  cout<<reverse(b)<<endl;

  return (0);
}
```

Compiler

```cpp
#include <iostream>
using namespace std;

int reverse (int x) {
  return (-x);
}

float reverse (float x) {
  return (-x);
}

int main() {
  int a = 10;
  float b=15.4;
  cout<<reverse(a)<<endl;
  cout<<reverse(b)<<endl;

  return (0);
}
```

Multiple arguments in function templates

```
template<typename T>
T findLargest( T x, T y, T z ) {
   T largest;
   largest = (x < y)? y : x;
   largest = (largest < z)? z: largest;

   return largest;
}
```

- `x`, `y`, `z`, and `largest` may be of any type for which the < operator is defined.
  - The = operator is at least default defined.
- They must all be of the same type because they are all defined to be the same type `T`.

```
( E1 ) ? E2 : E3;
```

- If `E1` is true, `E2` is evaluated.
- If `E1` is false, `E3` is evaluated.

- Use of this conditional or ternary operator can make code quite difficult to read if it’s layered or the individual expressions are quite complex.

```
class PhoneCall {
  friend ostream& operator<<( ostream&,const PhoneCall& );
  private:
      int minutes;
  public:
      PhoneCall(int = 0);
      bool operator<(const PhoneCall&);
};
ostream& operator<<(ostream& out, const PhoneCall& p) {
    out << "Phone call lasted " << p.minutes << " minutes" << endl;
    return out;
}
PhoneCall::PhoneCall(int ct) {
    minutes = ct;
}
bool PhoneCall::operator<(const PhoneCall& p) {
    bool less = (minutes < p.minutes)? true: false;
        return less;
}
```

Overloaded operator<

```
int main() {
    double a, b, c;

    cout << "Input values a b c: ";
    cin >> a >> b >> c;

    cout << "The largest double : " << findLargest( a, b, c ) << endl;

    PhoneCall call1(5);
    PhoneCall call2(8);
    PhoneCall call3(12);

    cout << "The longest call : " << findLargest( call1, call2, call3 ) ;
}
```

Overloading function templates

- You can overload function templates only when each version takes a different argument list, allowing the compiler to distinguish between them.

```
template<typename T>
void invert(T &x, T &y)
{
    T tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
template<typename T>
void invert( T &x )
{
    x = -x;
}
```

- So this is okay.

Mixed template/non template function arguments

```
template<typename T>
void repeatValue( T val, int num ){
    for( int i=0; i<num; i++ )
        cout << val << " ";
    cout << endl;
}
```

- The operator << has to be defined for all data types this template is to be used with.

Multiple generic types …

- As mentioned earlier, this is also allowed.

```
template< typename T, typename U >
void displayAndCompare(T val1, U val2) {
    cout << "val1=" << val1 << ", val2=" << val2 << endl;
    if(val1 < val2)
        cout << "The second one is larger." << endl;
    else if(val1 == val2)
        cout << "They are the same." << endl;
    else
        cout << "The first one is larger." << endl;
}
```

- To use this, == and < would need to be overloaded with the types we used it with.

```cpp
class PhoneCall {
    friend ostream& operator<<(ostream&, const PhoneCall&);
    private:
        int minutes;
    public:
        PhoneCall(int = 0);
        bool operator<( const PhoneCall & );  // PhoneCall < PhoneCall
        bool operator<( int n );              // PhoneCall < int
        bool operator==( const PhoneCall & ); // PhoneCall == PhoneCall
        bool operator==( int n );             // PhoneCall == int
};

ostream& operator<<( ostream& ost, const PhoneCall& p ) {
    ost << p.minutes;
    return ost;
}

PhoneCall::PhoneCall(int value) : minutes(value){}

bool PhoneCall::operator<( const PhoneCall& p ) {
    return ( (minutes < p.minutes)? true: false ); }

bool PhoneCall::operator<( int m ) {
    return ( (minutes < m)? true: false ); }

bool PhoneCall::operator==( const PhoneCall& p ) {
    return ( (minutes == p.minutes) ? true: false ); }

bool PhoneCall::operator==( int m ) {
    return ( (minutes == m)? true: false ); }
```

```
int main() {
    double a = 3.8, b=4.5, c=6.825;
    float x = 3.1415;
    displayAndCompare( a, b );
    displayAndCompare( a, x );

    PhoneCall call1( 5 );
    PhoneCall call2( 8 );

    displayAndCompare( call1, call2 );
    displayAndCompare( call1, 5 );
}
```

```
val1=3.8, val2=4.5
The second one is larger.
val1=3.8, val2=3.1415
The first one is larger.
val1=5, val2=8
The second one is larger.
val1=5, val2=5
They are the same.
```

- When defining a function template with several generic types, you need to make sure that all relevant operators are overloaded, for all data types and all used combinations.
- You may not have anticipated some combinations and that would result in compilation errors.

Explicit type specification…

- When you call a template function, the arguments to the function determine the types to be used based on what is given, so it's implicit typing.
- To override a deduced type when calling a function template you can explicitly specify a type name …

```
someFunction<char>(someArgument);
```

- This is useful when:
  - At least one of the types you need to generate in the function is not an argument, so otherwise won't be able to be deduced.
  - You want to limit the number of functions generated from a template and are able to readily cast between related types.

```cpp
#include <iostream>
using namespace std;

template <typename T>
T reverse (T x)
{
  return (-x);
}

int main()
{
  int a= 10;
  float b = 15.4;

  cout<<reverse(a)<<endl;
  cout<<reverse<int>(b)<<endl;

  return (0);
}
```

Compiler

```cpp
#include <iostream>
using namespace std;

int reverse ( int x )
{
  return (-x);
}


int main()
{
  int a= 10;
  float b=15.4;
  cout<<reverse(a)<<endl;
  cout<<reverse(b)<<endl;

  return (0);
}
```

```
-10
-15
```

Default template arguments

- C++ functions can have default arguments, so that if an argument isn't provided we have a value to use.
- From C++11 it's possible to provide default arguments for function templates, and for class templates too.
- Here goes an example from the textbook …
  - It uses a second type parameter, `F`, to specify the type of a callable object and a function parameter `f` bound to a callable object.
  - The default is a library function-object template `less`.

```
template <typename T, typename F=std::less<T>>
int compare(const T &v1, const T &v2, F f=F())
{
      if (f(v1, v2)) return -1;
      if (f(v2, v1)) return 1;
      return 0;
}
```

```
#include <iostream>
#include <functional>
using namespace std;

template <typename T, typename F=std::less<T>>
int compare(const T &v1, const T &v2, F f=F())
{
        if (f(v1, v2)) return -1;
        if (f(v2, v1)) return 1;
        return 0;
}

int main()
{
        cout << compare(10,0) << endl;
        cout << compare(0,10) << end;
        cout << compare(10,10) << endl;

        return 0;
}
```

Function objects

- Objects that are instances of classes that have an overloaded `operator()`, the function call operator, are considered to be function objects.
- We can use them as if they were functions, so it looks like we call an object.

Non-type template parameters

(also called value parameters)

- Non-type template parameters represent values rather than types, and instantiation results in a version of a template with a specific fixed value.
- Those values must be constant expressions, which the compiler can evaluate at compile time.
- Here goes an example from the textbook.

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
      return strcmp(p1, p2);
}
```

`compare("hi", "mum");` produces

```
int compare(const char (&p1)[3], const char (&p2)[4])
```

Compile time computation

- There are a few reasons why it can be helpful to do compile time calculations:
  - Efficiency: Pre-calculation of a value, often a size.
  - Type-safety: Computing a type at compile time.
  - Simplify concurrency: You can’t have a race condition on a constant.
- Why not just fix them and calculate at programming time?
  - They aren’t necessarily the same for each compilation.
  - We might have a large table of values that we are going to need to use over and over once a program is running, but explicitly entering the values could use up a lot of space.

constexpr (C++)

01/28/2020 • 5 minutes to read • C C M M K +3

The keyword `constexpr` was introduced in C++11 and improved in C++14. It means *constant expression*. Like `const`, it can be applied to variables: A compiler error is raised when any code attempts to modify the value. Unlike `const`, `constexpr` can also be applied to functions and class constructors. `constexpr` indicates that the value, or return value, is constant and, where possible, is computed at compile time.

A `constexpr` integral value can be used wherever a const integer is required, such as in template arguments and array declarations. And when a value is computed at compile time instead of run time, it helps your program run faster and use less memory.

https://docs.microsoft.com/en-us/cpp/cpp/constexpr-cpp?view=msvc-160

```cpp
#include <iostream>
using namespace std;

constexpr int factorial(int n) {
    return (n ? (n * factorial(n - 1)) : 1);
}

constexpr int f10 = factorial(10);

int main() {
    cout << f10 << endl;
    return 0;
}
```

Based on `https://rosettacode.org/wiki/Compile-time_calculation`

- The `constexpr` is C++'s way of requesting that we evaluate something at compile time.

A note on templates

- A *template implies an interface*.
  - That is, the **template** keyword says “I’ll take any type,” …
  - … however the code in a template definition requires that certain operators and member functions be supported—that’s the interface.
  - So, a template definition is really saying, “I’ll take any type that supports this interface.”