Generic Programming

Templates and Function objects
Motivation

- Reuse code!

- Proprocessor macros: A partial solution (since cpp, the compiler preprocessor, does not know C), e.g.

  ```
  #define MAX(_a,_b) ((_a)>(_b) ? (_a) : (_b))
  ```

- Possible solution: using callbacks, e.g.

  ```
  void qsort (void *base, size_t n, size_t size,
  int (*cmp)(const void *, const void *));
  Set SetCreate(bool cmp(Element,Element),
  Element cpy(Element),
  void fre(Element));
  ```

- C++ solution: Templates (for functions and classes)
C++ provides a super macro mechanism identified by the keyword `template`.

Templates can be thought of as MACROs for `functions` and `classes`, but they are controlled by the compiler (and not by `cpp`).

Template arguments are either `class-types` or `const-expressions` of a fixed type.

Arguments that are passed to template functions are checked for type-mismatch.
Function Templates

Declaration

template <class T>
T max(T a, T b)
{  return a>b?a:b;
}

Usage

int main()
{  int a = 3,  b = 4;
  char c = ‘a’, d = ‘b’;

  cout << max(a,b);  // o.k.
  cout << max(c,d);  // o.k.
  cout << max(a,c);  // Error!!
  return 0;
}
Resolving Function Names

- **Overloading** for functions of the same name is done in this order:
  - Exact match among (non-template) functions.
  - Exact match that can be generated from a function template.
  - Best match, using (in that order):
    - *Trivial* conversions (which are considered to be exact match)
    - Promotions (e.g., short to int)
    - *Standard* conversions (e.g., int to double)
    - *User-defined* conversions (e.g., double to complex)

Beware of “dark corners”!

- For each call to a function template, the compiler will **generate an actual function** with the appropriate prototype.
- Usually, every *template-argument* specified in the *template-argument-list* should be used in the *argument-list* of a function template.
int main()
{
    int a = 3, b = 4;
    char c = 'a', d = 'b';
    typedef char const *String;
    String s = "abc", t = "xyz";
    cout << max(a,b);    // o.k.
    cout << max(c,d);    // o.k.
    cout << max(s,t);    // special
    return 0;
}

// Explicit template specialization
template<> String max<String>(String a, String b)
{ return strcmp(a,b) < 0 ? b : a; }

Both implementations have the same effect. What happens if we do both?
Setting an exception to a general template:
Class Templates

- Class templates are used to define *generic classes*.

  ```cpp
  template<class E, int size> class Buffer;
  ```

- An **actual class** is created only when an actual object is defined, using the class template with actual parameters.

  ```cpp
  Buffer<char *, 1024> buf_chrp;
  Buffer<int, 1024> buf_int;
  ```

- Two *template-classes* refer to the same class if their *template-names* are identical (typedefs included) and their arguments have identical types and values.

  ```cpp
  typedef char* String;
  Buffer<String, 2*512> buf_str;
  ```
template <class T>
class Vector {
    int size_;  
T  *data;
    static int set(int);  // Set the size only if it
public:  // within the right range
    Vector(int sz = 16): 
        size_(set(sz)),data(new T[size_]){}
    Vector(const Vector& source);  // copy Ctor
    ~Vector() { delete[] data;}
    Vector& operator=(const Vector& source);
    int size() const { return size_;}
    T& operator[](int indx);
};

T must have an argumentless Ctor, otherwise new[] will fail!
A member function of a template class is automatically a template function.

```cpp
template <class T>
int Vector<T>::set(int sz) {
    if (sz < 1) error("wrong size");
    return sz;           // Exceptions :-(
}

template <class T>
T& Vector<T>::operator[](int index) {
    if (index < 0 || index >= size_)
        error("out of bound");    // Exceptions :-(
    return data[index];
}
```

Overloading the index operator
```c
#include <math.h>
#include "Vector.h"

int main () {
    Vector<double> arr_d(4), arr_e = arr_d; // ...
    for (int i = 0; i < arr_d.size(); ++i)
        arr_d[i] = sqrt(i); // ...
    Vector<Vector<double> > vvd(42); // row size is 16
    vvd[0] = arr_e; vvd[1] = arr_d;
}
```

Note the space!

In each level, what argument does the constructor get?
Overloading operator []

The following will not compile:

```cpp
{ Vector<double> arr1(4), arr2(4);
  // . . .
  const Vector<double> carr(arr1);
}
```

operator[] is not const

The following prototype is a Trojan-Horse:

```cpp
T& operator[](int indx) const;
```

Why?!

Changing the prototype to

```cpp
const T& operator[](int indx) const;
```

will cause a different problem:

```cpp
arr2[2] = 1;
```

cannot assign to a const object

Q: What is the problem here?! (A: Trying to achieve two goals at once.)
Overloading operator[] (cont.)

What we really need is **two** versions of `operator[]`:

```
T const& operator[](int indx) const;
T& operator[](int indx);
```

Avoiding code duplication, we implement the non-const by delegation to the const one:

```
T& operator[](int indx) {
    return const_cast<T&>
        (const_cast<T const&>
         (this)->operator[](indx));
}
```

This way we are able to forward the semantics of constants to sub-objects which are not syntactically constants.

This is a part of what is known as **const-correctness**.
In contrast with member functions, *friend functions* of a template are NOT implicitly template functions.

```cpp
template <class T>
class task {
// ...
    friend void next_task();
    friend task<T>* do_task(task<T>*);
    friend task* print_task(task*); // O.K.
};
```

- There is a *unique function* `next_task()`, a friend of all task classes.
- Each task class has an appropriate *friend function* `do_task()`.
- In `print_task()`, `task` is `task<T>`. (*A syntax error in Stroustrup, 1991 edition.*)
Programming Templates: Advice

• A template compilation is a two-stage process
  1. The template is checked for basic errors
  2. Each instantiation (with actual parameters) is compiled and converted to machine code
• The Second stage may encounter many errors that could not have been found at the first stage — especially when the template depends on other templates
• Suggestion: If a template is complicated, start by writing it for a specific type, until compilation succeeds

```cpp
// template <typename T>
typedef double T; // Remove after compilation succeeds
class Array {
    int size; T *a;
public:
    Array(int sz); ...
};
```
Motivation: Reusability of quality code

Common data structures:
- Containers (storage, iterators, algorithms)
- Strings (supporting different character types, no need for raw C-style character arrays and pointers)

C++ tool: STL (Standard Template Library)

Examples of containers: vector, deque, list, set, map, array, forward_list, unordered_map, ...

Iterators:
- Can be const and/or reverse
- Can serve streams: istream_iterator, ostream_iterator

Algorithms: for_each, find, copy, replace, remove, reverse, sort, merge, ...
STL Containers

- **Sequential:**
  
  ```
  template <class T, class Allocator = allocator<T> >
  class vector { /* ... */ };  
  ```

- **Associative:**
  
  ```
  template <class Key, class T,
  class Compare = less<Key>,
  class Allocator = allocator<pair<const Key, T> > >
  class map { /* ... */ };  
  ```

- **Huge interface (breaking the “minimal and complete” principle)**
  
  It’s unreasonable to write many implementations of `vector`, each having a different interface – even if each one is minimal and complete (in some context) – which will overlap.
When we don’t know the element type:

```cpp
for (typename std::vector<double>::reverse_iterator i = v.rbegin(); i != rend(); ++i)
```

Using `auto` (type defined automatically by the compiler):

```cpp
for (auto i = v.rbegin(); i != v.rend(); ++i)
```

This is stronger than `typedef` but fragile when implicit types are used and changed

Range-based `for` statements become simpler:

```cpp
for (double &x: v) { ... }  // Iterates forward
```
Alias Declarations may replace `typedef` using `vecPrsnIt = std::vector<Person>::iterator;` `typedef std::vector<Person>::iterator vecPrsnIt;`

Alias Declarations are a bit clearer than `typedef` using `CmpPtrF = bool (*)(Person const&, Person const&);` `typedef bool (*CmpPtrF)(Person const&, Person const&);`

Alias Declarations can be defined as `templates` `template <class T> // Use: CmpPtrF<Person> f;` `using CmpPtrF = bool (*)(T const&, T const&);` `template <class T> // A typedef must be nested in a class` `struct CmpF { // Use: typename CmpF<Person>::Ptr f;` `{ typedef bool (*Ptr)(T const&, T const&);` `};`
STL Iterators

- Iterators are an abstraction of pointers, and they are safer.
- Operators: !=, *, ->, ++, -- (and possibly more)
- Modifications to a container may invalidate its iterators
- Types of iterators are (in descending specialization):
  - Random access
  - Bidirectional
  - Forward (includes Reverse) (no operator--())
  - Input/Output (Output is a bit “weaker”)
There are SO MANY of them

Possibly more efficient than loops, and reduce considerably the need to implement them

Operate on half-open ranges [first, last)
STL Algorithm Example: find

- Searching for a value:

  ```cpp
template<class InputIterator, class T>
  InputIterator find(InputIterator first, 
                     InputIterator last, 
                     const T& value);
  ```

- Searching for a property:

  ```cpp
template<class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first, 
                        InputIterator last, 
                        Predicate pred);
  ```

- `Predicate` is a function pointer or a function object which is passed by value
Function Objects

- Purpose: Using an object as a function
- A function object belongs to a class which overloads operator() and usually has little or nothing more

```cpp
class close_enough {
    public:
        close_enough (int val, int distance) : v(val), d(distance) {/* d >= 0 */}
        bool operator()(int num) const { return v - d < num && num < v + d; }
    private:
        int v, d;
};
```

- A function object is a light-weight object and is usually used as a value (and not by-reference).
- Allows inlining (when operator() is quick) for boosting speed
- Sample use: `close_enough(k)` to find if $1325 \leq k \leq 1355$.
- Another example:

```cpp
auto it = find_if (v.begin(), v.end(),
                    close_enough (1340, 15));
```
**Function Object: A Complex Example**

**From C Library:**

```c
void qsort (void *base, size_t n, size_t size,
            int (*cmp) (const void *, const void *));
```

**From C++ Library:**

```cpp
template<class RandAccIter, class Cmpr>
void sort (RandAccIter first, RandAccIter last, Cmpr cmp);

template <class T> struct less { // From C++ library
    bool operator() (const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};
```

**User code:**

```cpp
std::vector<Person> v; // Populate v
sort (v.begin(), v.end(), less<Person>());
```
Smart Pointers

- The class `MemManager`, used for showing the overloading of the operator `*` and `->`, is a simple version of the “Smart Pointer” concept.

- **Purpose:** Easing the management of memory.

- **Two basic types (C++11) of smart pointers:**
  - `unique_ptr`: each piece of memory has a unique owner.
  - `shared_ptr`: pieces of memory can have many owners.
  - `weak_ptr`: similar to `shared_ptr` but without ownership (existence may need to be tested).
A Container Example

#include <iostream>
 using std::cin;
 using std::cout;
 using std::endl;

#include <string>
 using std::string;

#include <vector>
 using std::vector;

#include <list>
 using std::list;

#include <deque>
 using std::deque;

typedef int lmnt;      // or string
typedef deque<lmnt> container; // or list or vector
typedef container::iterator iterator;

Introduction to Systems Programming
int main()
{
    int const len = 10;  // May be initialized by len(10);
    lmnt  l[len] = { 9, 15, 12, -5, 1};
    // { "abc", "xyz", "pqr", "st"};

    container  cn0, cn1(10);
    cn0.push_back (l[0]);
    cn0.push_back (l[1]);
    cn0.push_front(l[2]);  // Illegal for "vector"
    cn0.push_front(l[3]);  // Illegal for "vector"
    cn0.push_back (l[3]);

    for (iterator i = cn0.begin(); i != cn0.end(); ++i){
        cout << *i << endl;
    }

    return 0;
}
int main() {
    int const len(10); // Same as initialization: int const len = 10;
    lmnt  l[len] = {9, 15, 12, -5, 1},
               m[]{1,2,3,4,5}, n[len]{6,7,8,9};
    container cn0 = {1,2,3}, cn1({2,3,4}), cn2{5,6,7};

    for (int n: {9,8,7}) { // A range-based for loop
        cn0.push_back(n);
    }

    for (int n: m) {
        cn0.push_back(n);
    }

    for (int &m: cn0){ // Reference allows modifications
        m += 2;
    }

    for (int m: cn0){ // The modified elements
        cout << m << ','; // Prints: 3,4,5,11,10,9,3,4,5,6,7,
    }

    return 0;
}
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main() {
    istream_iterator<int> is(cin);
    ostream_iterator<int> os(cout);
    for (int len = *is++; len-- > 0; ++is, ++os) {
        *os = *is;
        cout << ' ';
    }
    // An extra int may be read! However,
    if (cin.fail()) cin.clear(); // printing int-as-string buys nothing
    istream_iterator<string> ist(cin);
    istream_iterator<string> end_of_stream;
    ostream_iterator<string> ost(cout);
    string space =", "; // May be initialized by string space("", ");
    for (; ist != end_of_stream || (*ost = "\nFIN\n", 0); ++ist, ++ost) {
        *ost = *ist;
        +++ost = space;
    }
    return 0;
}
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

typedef int
lmnt;
typedef list<lmnt>
container;
typedef container::iterator
iterator;
lmnt const target(5);

int main() {
    istream_iterator<lmnt> is(cin), eos;
    container bucket;

    copy(is, eos, back_inserter(bucket));
    iterator it = find(bucket.begin(), bucket.end(), target);
    ostream_iterator<int> os(cout, ", ");
    os = copy(it, bucket.end(), os);
    cout << endl;
}