Chapter 8

The C++ Programming Language

Generic Programming (templates) and STL
The Principle of Little Numbers

0 > 1: What you don't know cannot confuse you. Use *Information hiding, encapsulation, table driven programming, modularity, etc.*

1 > 2: If your code occurs twice in a program, then it occurs at least once too many.

Software is constantly subject to change. As changes occur, maintaining two, almost identical, parts will become a nightmare.

For (ever) {
    Write Code;
    While (exist(similar parts)) {
        Extract identical portion;
        Rewrite code;
    }
}
When we hit code duplication in C

- The C library solution

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

- This is just a small scale of our Set ADT

```c
Set SetCreate(bool cmp(Element, Element),
              Element cpy(Element),
              void fre(Element));
```
When we hit code duplication in C

• But it doesn’t scale down!

```c
void const *max(void const *a, void const *b,
                 int (*cmp)(void const *, void const *));
```

• The standard MACRO is only a partial solution

```c
#define MAX(a, b) (((a) > (b) ? (a) : (b))
```

– We need `>` to be already defined for the objects
– We cannot use function/operator overloading in C
When we hit code duplication in C

The preprocessor at our rescue (simplistic)

```c
#ifndef T_GREATER
    #define T_GREATER(x, y) ((x) > (y))
#endif

#ifdef FUNC_NAME
    #define F_NAME(T) FUNC_NAME
#else
    #define F_NAME(T) T##_MAX /* C preprocessor idiosyncrasies */
#endif

#define DECLARE_MAX(T)  
    T F_NAME(T)(T a, T b) 
    { return T_GREATER((a), (b)) ? (a) : (b); }
DECLARE_MAX(TYPE)

#undef FUNC_NAME
#undef F_NAME
#undef T_GREATER
#undef TYPE
```

When we hit code duplication in C

The preprocessor at our rescue (simplistic)

- **Usage**

```c
#include <stdio.h>
#include <string.h>

#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main()
{   printf("int_MAX(5,9) = %d\n", int_MAX(5,9));
    printf("MAX(5,9.1) = %d\nMAX(5,4.1) = %f\n", MAX(5,9.1), MAX(5,4.1));
    printf("ldbl_max(5,4.02) = %Lf\n", ldbl_max(5,4.02));
    printf("String_MAX("abcd","xy") = %s\n", String_MAX("abcd","xy"));
    return 0;
}
```

When we hit code duplication in C

The preprocessor at our rescue (simplistic)

- **Usage**

```c
#include <stdio.h>
#include <string.h>

#define TYPE int
#include "MAX.h"

#define TYPE long double
#define FUNC_NAME ldbl_max
#include "MAX.h"

typedef char *String;
#define TYPE String
#define T_GREATER(x,y) (strcmp((x),(y)) > 0)
#include "MAX.h"

#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main()
{   printf("int_MAX(5,9) = %d\n", int_MAX(5,9));
    printf("MAX(5,9.1) = %d\nMAX(5,4.1) = %f\n", MAX(5,9.1), MAX(5,4.1));
    printf("ldbl_max(5,4.02) = %Lf\n", ldbl_max(5,4.02));
    printf("String_MAX("abcd","xy") = %s\n", String_MAX("abcd","xy"));
    return 0;
}
```
When we hit code duplication in C
The preprocessor at our rescue (simplistic)

- **Preprocessor output** (many spaces were removed)

```c
# 3 "main.c" 2
# 1 "MAX.h" 1
# 15 "MAX.h"
    int int_MAX(int a, int b) { return (((a)) > ((b)))?(a):(b);}    
# 7 "main.c" 2
# 1 "MAX.h" 1
# 15 "MAX.h"
    long double ldbl_max(long double a, long double b) {return (((a))>((b)))?(a):(b);}    
# 11 "main.c" 2
typedef char *String;
# 1 "MAX.h" 1
# 15 "MAX.h"
    String String_MAX(String a, String b) { return (strcmp(((a)),((b))) > 0)?(a):(b);}    
# 16 "main.c" 2
int main()
{ printf("int_MAX(5,9) = %d\n", int_MAX(5,9));
    printf("MAX(5,9.1) = %d\n", ((5) > (9.1) ? (5) : (9.1)));
    printf("ldbl_max(5.01,4.02) = %Lf\n", ldbl_max(5.01,4.02));
    printf("String_MAX("abcd","xy") = %s\n", String_MAX("abcd","xy"));
    return 0;
}
```

When we hit code duplication in C
The preprocessor at our rescue (simplistic)

- **Preprocessor output** (many spaces were removed)
When we hit code duplication in C
The preprocessor at our rescue (simplistic)

• Program output

```
int_MAX(5,9) = 9
MAX(5,9.1) = 858993459
ldbl_max(5.01,4.02) = 5.010000
String_MAX("abcd","xy") = xy
```
TEMPLATES

• **C++** provides a *super macro mechanism* that is identified by the keyword `template`.

• **Templates** can be thought of as MACROs for *functions* and *classes*, that need not be explicitly called by the programmer.

• **Templates** are controlled by the *compiler* (and *not* by the pre-processor).

• **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**

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

**Usage**

```cpp
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;
}
```

Defining additional function `int max(int,int)` will completely change behaviour.

Operator `>` should be defined for class T.
Function Templates (Cont.)

- For **each call** to a function template, the compiler will try to **generate an actual function** with the appropriate prototype.

- **Overloading** resolution for functions of the same name is done in this order:
  - Look for an **exact** match among (non-template) functions.
  - Look for an **exact** match that can be generated from a function template.
  - Look for the **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**)

- Usually, every **template-argument** specified in the **template-argument-list** should be used in the **argument-list** of a function template.
Function Templates (Cont.)

Special Needs

```c++
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;
}
```

Implementation A

// Regular Function
String max(String a, String b)
{ return strcmp(a,b) < 0 ? b : a; }

Implementation B

// Explicit Template Specialization
```c++
template<> String max<String>(String a, String b)
{ return strcmp(a,b) < 0 ? b : a; }
```
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;
  ```
A Class Template Example

template <class T>
class Vector {
    int size_; 
    T  *data;
    static int set(int);  // correctly set the size
public:
    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);
};
A member function of a template class is implicitly a template function.

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

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
A Class Template Example (Cont.)

```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;
}
```
Overloading `operator[]`

The following will not compile:

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

The following prototype is a Trojan-Horse:

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

Changing the prototype to

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

will bring a different problem:

```cpp
arr2[2] = 1;
```
Overloading `operator[]` (Cont.)

We really need two versions of `operator[]`:

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

`a[2]` will activate the appropriate function according to the `const-ness` of `a`!

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

```cpp
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`.

No distinction is made related to `read` or `write` operation used. For that – you need `proxy classes` (More effective C++, S. Meyers).
A Class Template Example (Cont.)

In contrast to 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()`, friend of all task classes.
- Each task class has an appropriate friend function `do_task()`.
- In `print_task()`, `task` is `task<T>`.

*(was syntax error in BS 2nd ed.1991)*
Programming Templates (advice)

• A template compilation is a two-stage process
  • First, the template is processed, checked for basic errors
  • Second, for each instantiation (with actual parameters) the class/function is finally translated to machine lang.
  • 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
  • So far, such error messages are hard to decrypt 😞

• Suggestion: If your template code is not simple, 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); ... };
```
High Level Programming
How C++ makes programming easier
(If you know C++ :-) 

• The main strategy of quality programming is Reuse, Reuse, Reuse …
• One of the more common tasks in software is the usage of Containers
• Using containers requires the usage of Iterators and Algorithms
• C++ provides in its standard library self resizable containers, iterators and common algorithms. It is made in the style of Generic Programming and is implemented by using C++ Templates.
• In short, this is now known as STL, the Standard Template Library
High Level Programming  (cont.)

• A special type of container is **string**, that may contain different types of character as needed. It eliminates the need for using raw C-style character arrays and pointers.

• Some of the containers are  
  
  vector, deque, list, set, map, array, forward_list, unordered_map  
  
  (C++11)

• **Iterators may be const and/or reverse.**  
  There are also  
  
  istream_iterator, ostream_iterator

• **Among the algorithms there are**  
  
  for_each, find, copy, replace, remove, reverse, sort, merge
STL Containers

• **Sequential containers**
  template <class T, class Allocator = allocator<T> >
  class vector { /* ... */ };

• **Associative containers**
  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 vectors, each with a different interface –
    even if it is minimal and complete (in some context) – which will overlap.
STL Containers  (Cont.)

• C++11 solved the nearly unmanageable nested-types problem (which typedef only marginally solves):

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

becomes

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

• This is a bit stronger than a typedef, but may be more fragile when types are changed (because it is implicit).

• Range-based for statement may simplify it further

```cpp
for (double &x: v) // Iterates forward
```
C++11: Alias Declarations

**Alias Declarations may replace** `typedef`

```cpp
using vecPrsnIt = std::vector<Person> iterator;
typedef std::vector<Person> iterator vecPrsnIt;
```

**Alias Declarations a bit clearer than** `typedef`

```cpp
using CmpPtrF = bool (*)(Person const&, Person const&);
typedef bool (*)(CmpPtrF)(Person const&, Person const&);
```

**Alias Declarations can be defined as** `templates`

```cpp
template <class T> // Use: CmpPtrF<Person> f;
using CmpPtrF = bool (*)(T const&, T const&);

template <class T>
struct CmpF {
    // Use: typename CmpF<Person>::Ptr f;
    typedef bool (*)(Ptr)(T const&, T const&);
};
```
STL Iterators

• They are abstraction of pointers, and safer.
  • They have operators  

  (and may be more)

• Modifications of a container may invalidate its iterators.

• Types of iterators are (in descending specialization):
  • Random access
  • Bidirectional
  • Forward (includes Reverse) (operator ––() is not implemented)
  • Input/Output (Output is a bit “weaker”)
STL Algorithms

- Operate on half-open ranges \([\text{first}, \text{last})\)
  - They considerably reduce the need to write loops
- They are more efficient than loops
  - Brian Malloy (http://www.cs.clemson.edu/~malloy/):
    searching on a big vector - O0 (O3) (results are rounded):
    - Indexed loop  - 150sec (27sec)
    - Iterator loop  - 50sec (26sec)
    - `find()` - 35sec (17sec)
- There are so many of them.
STL Algorithms Example: Find

- **Search for a value**
  
  ```cpp
  template<class InputIterator, class T>
  InputIterator find(InputIterator first, InputIterator last, const T& value);
  ```

- **Search 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

• A function object belongs to a class that overloads `operator()` and (usually) little 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)

```cpp
auto it = find_if(v.begin(), v.end(),
                  close_enough(1340, 15));
```
Function Objects: Back to First 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>());
```
Function Objects (cont.)

- Function objects allow inlining (when `operator()` is short enough), so can be more efficient than function pointers.

Lambda (Λ) Expressions (C++11)

What if, in a previous example, an object close_enough is used only once? Is it worth the efforts of defining a class? It is just a matter of wasting efforts, since function objects are very useful in many situations, but often used only once.

In such cases, lambda expressions (originated in functional programming) come to the rescue. Here is how it can be used:

```cpp
auto it = find_if(v.begin(), v.end(), close_enough(1340, 15));
```

becomes

```cpp
auto it = find_if(v.begin(), v.end(),
                  [] (int num) { return 1325 < num && num < 1355; });
```

- The compiler will create a function-object class (called closure class).
- At runtime, there will be created a function-object (called closure).
- A closure may contain objects or values from the scope of the Λ-expression.
- The rest – go study (אינא Whip-זציל וואדך).
Smart Pointers

In introducing the overloading the operators *, -> we defined the class MemManager. Actually, it was a simplistic version of what is known as a Smart Pointer. Various smart pointers ease tasks of managing memory.

C++11 provides, basically, two types of smart pointers:

- **unique_ptr** – it manages memory that should have a unique owner.
  - When ownership should not be transferred, it’s like Boost::scoped_ptr.

- **shared_ptr** – it manages memory that is shared among some owners.
  - **weak_ptr** – is similar, without ownership, existence may need to be tested.
#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;
int main()
{
    int const len = 10; // May be initialized by len(10);
    int const 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[], 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;
}
A Stream-Iterator Example

```cpp
#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;
}
```

Possible input for 3 integers:

```
3 9 22 17
```
An Algorithm Example

```cpp
#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;
}
```