249

Chapter 8

The C++ Programming Language

Generic Programming (templates)

250

The Principle of Little Numbers

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

1 > 2: If the same thing occurs twice in the code, then it occurs 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;
   }
}
```

251

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.

252

Function Templates

Declaration

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

operator> should be defined for **class** T

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;
}
```

Defining additional function
`int max(int,int)`
will prevent the compilation error.

253

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.

254

Class Templates

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

```
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.

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

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

```
typedef char *   string;
Buffer<string, 2*512> buf_str;
```


255

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);
};
```

T must have an argumentless Ctor

256

A Class Template Example (Cont.)

A member function of a template class is implicitly a template function.

```
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

257

A Class Template Example (Cont.)

```
#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;
  vvd[0] = arr_e;  vvd[1] = arr_d;
}
```

258

Overloading `operator[]`

The following will not compile:

```
{ Vector<double> arr1(4), arr2(4);
  // . . .
  const Vector<double> carr(arr1);
  arr2[2] = carr[2] + 1;
}
```

`operator[]` is not `const`

The following prototype is a Trojan-Horse:

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

Changing the prototype to

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

will bring a different problem:

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

cannot assign to a `const` object

259

Overloading `operator[]` (Cont.)

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

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

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

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

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.

260

A Class Template Example (Cont.)

In contrast to member functions, *friend functions* of a template are **NOT** implicitly template functions.

```
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 error in BS 2nd ed.1991)