COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Generic Code

- C++ supports generic programming
  - This is writing code which works with many different types
- It gives some benefits of weak typing
- With the safety of strong typing
- The standard library uses it extensively

Finding an int in a vector

- We could write the following code

```
vector<int>::size_type
find(const vector<int> &v,
                    const int &value) {
    for (vector<int>::size_type i = 0;
            i < v.size(); ++i)
        if (v[i] == value)
            return i;
    return v.size();
}
```

Finding a string in a vector

- Very similar code

```
vector<string>::size_type
find(const vector<string> &v,
                    const string &value) {
    for (vector<string>::size_type i = 0;
            i < v.size(); ++i)
        if (v[i] == value)
            return i;
    return v.size();
}
```

Silly to write it twice

- Writing the same code twice is bad
- It will get out of sync
  - Changes in one will be forgotten in the other
- It's a waste of time
- C++ provides a solution
- Can write one "template function" which works for any vector

Template Functions

```
template<typename T>
typename vector<T>::size_type
find(const vector<T> &v,
                    const T &value)
{
    for (typename vector<T>::size_type
                i = 0; i < v.size(); ++i)
        if (v[i] == value)
            return i;
    return v.size();
}
```

Typename...

- template<typename T>
  - Makes the function a template
  - T will be replaced with a type
- typename vector<T>::size_type
  - T is unknown when compiling
  - vector<T>::size_type is unknown
  - Could be a member or a type
  - The compiler assumes a member
  - typename specifies it's a type

Calling Template Functions

- Called the same as normal functions
- find(vi, 5);
  - vi is a vector<int>
  - Will instantiate the template
  - Basically replacing the T's with int's

Another Template Function

- Here's another template

```
template <typename T>
T sum(const vector<T> &v)
{
    T result = 0;
    for (typename vector<T>::size_type
            i = 0; i < v.size(); ++i)
        result += v[i];
    return result;
}
```

Using the Template

- What would the following do?

```
int main() {
    vector<int> v;
    v.push_back(1);
    v.push_back(10);
    v.push_back(5);
    cout << sum(v) << endl;
}
```

Using the Template II

- How about this code?

```
int main() {
    vector< vector<int> > v;
    // fill v with data
    cout << sum(v) << endl;
}
```

Common Behaviour

- Templates assume operator behaviour
- Only types with the operators will work
  - vector has no += operator
- Operators must behave appropriately
- The language can't enforce this

Using the Template III

- What about this:

```
int main() {
        vector<string> v;
        v.push_back("1");
        v.push_back("10");
        v.push_back("5");
        cout << sum(v) << endl;
}
```

A Real Problem

- It compiles but will crash when run
- The += operator isn't the problem
- The problem is:

```
    T result = 0;
```

- This is a much more serious problem
- There is no general solution
- Care is required when using template functions

Sequential Access

- The find() template function is an example of sequential access
  - Each item is used only after its prior item
- So we should rewrite it to use iterators
- We'll take the chance to reformat it too

Find with Iterators

```
template<typename T>
typename vector<T>::const_iterator
find(const vector<T> &v,
                          const T &value)
{
   typename vector<T>::const_iterator
                b = v.begin(), e = v.end();
   while ( (b != e) && (*b != value) )
         ++b;
   return b;
}
```

What About Lists?

- The find code would work for lists too

```
template<typename C, typename T>
typename C<T>::const_iterator
find(const C<T> &v, const T &value)
{ ... }
```

- However, this isn't allowed by C++
- So we need rewrite it for list<T>
- But code repetition is bad...

Iterators

- This is the main reason for iterators
- vector and list iterators
  - Sequential access operators
  - The code already uses iterators
  - We just need to use them directly

The Final find()

```
template <typename It, typename T>
It find(It begin, It end, const T &value) {
      while ( (begin != end) &&
                (*begin != value) )
            ++begin;
      return begin;
}
```

Using find()

- The caller code has to be changed
- vector<int> vi;
- find(vi.begin(), vi.end(), 5);
- list<string> ls;
- find(ls.begin(), ls.end(), "five");
- list<string> ls;
- list<string>::const_iterator b= ls.begin();
- find(++b, ls.end(), "five");

Iterator functions

- The standard library provides a find()
- It is exactly what we implemented
- This is a common idiom
- It allows the function to work with any container type (including your own)
  - The container just has to have iterators which support:
    - ++ for next
    - * for dereference
    - != for comparison

What about classes?

- We have a List class for ints
- We can make it general to any type

```
template<typename T>
class Node {
      T value;
      Node<T> *next;
      ...
};
```

Template classes

```
template<typename T>
class List {
      Node<T> *head;
      ...
};

template<typename T>
void List<T>::insert_at_front(T val) {
   head = new Node<T>(val, head);
}
```

Using template classes

- Same as how you have already seen
- Eg. vector is a template class

```
List<int> li;
li.insert_at_front(3);
li.insert_at_front(42);

List<string> ls;
ls.insert_at_front("3");
ls.insert_at_front("42");
```

Nested template classes

```
template<typename T>
class List {
    class Node {
        T value;
        Node *next;
        ...
    };
    Node *head;
    ...
};
```

- Previously: `Node<int> ni;`
- Now: `List<int>::Node ni;`

Template code

- Consider

```
template <typename T>
T add(T t1, T t2) {
    T result = t1 + t2;
    return result;
}
```

- Now you do

```
int i = 3, j = 9;
cout << add(i, j) << endl;
```

Instantiating templates

- The compiler instantiates the template
- This creates a new function

```
int add(int t1, int t2) {
    int result = t1 + t2;
    return result;
}
```

- To do this, compiler needs to know the function's code

Defining templated functions

- Normally a function definition is just

```
int add(int t1, int t2);
```

- But if it's templated, the definition is

```
template <typename T>
T add(T t1, T t2) {
    T result = t1 + t2;
    return result;
}
```

- Because the function code is needed for instantiation later

Template definitions

- Templated functions/classes aren't "real" functions/classes
- They're just the "outline" for real ones
- Thus code for templated functions/ classes must appear in header files
  - Inline with class definition or
  - External after class definition
- This is the ONLY time code is allowed in a header file
  - Don't #include .cc files