Comp151

Generic Programming and the STL

Example: Class Person

```
class Person
{
  public:
    void set_name(const string& n);
    void set_address(const string& adr);
    void set_email_address(const string& adr);
    string get_email_address() const;
    string get_name() const;
    string get_address() const;

  private:
    string name;
    string address;
    string email_address;
};
```

Example: Class Person_Container As An Array

```
class Person_Container
{
  public:
    Person_Container(int n) : MAX_SIZE(n), size(0)
         { array = new Person [MAX_SIZE]; }
    int size() const { return size; }
    void add_person(const Person& pers);
    Person get_person(int i) const;
    Person delete_person(int i);

  private:
    const int MAX_SIZE;
    Person* array;              // One-time pre-allocated storage
    int size;                   // Number of Persons actually stored
};
```

Container Class

Classes that maintain collections of objects are so common that they have been given a name: Container Classes.

- Let's write a program to maintain a collection of persons, and apply some operations on that collection.
- The operations on `Person_Container` can be:
  - member functions of the `Person_Container` class.
  - global functions that take a `Person_Container&` argument.
- Here we print mailing labels for all the persons, and send emails to invite them to our party.
- However, in the future we may want to reuse the `Person_Container` in a completely different application.
- Thus, we'll keep the class interface small, and we make the functions that we only need in the current application global.

Example: Operations on Person_Container

```
void print_mailing_labels(const Person_Container& pc) {
    for (int i = 0; i < pc.size(); ++i) {
        Person pers = pc.get_person(i);
        cout << pers.get_name() << endl;
        cout << pers.get_address() << endl;
        // ...
    }
}

void invite_to_party(const Person_Container& pc) {
    for (int i = 0; i < pc.size(); ++i) {
        Person p = pc.get_person(i);
        string command = "cat party.txt | mail ";
        command += p.get_name();
        system( command.c_str() );            // Send invitation emails
    }
}
```

Similar Code

- Note the *similarities* in both functions: they both set up a loop to do something for all persons in the container.
- We can expect that if we add more functions that do something with all persons, that these functions show the same similarities.
- So we could reuse one of the existing functions by copying it, and changing the name and the body of for-loop of the copy. In fact, that is what we did when we made the second function.
- However, code reuse by copying is often a <u>bad</u> idea. Why?

Array or Linked List?

- In some applications it is very convenient that we implement `Person_Container` with an array; the `get_person()` member function takes only $O(1)$ (constant) time, and we use that member function a lot.
- However, in other applications we may find that we frequently need to merge two `Person_Containers` into a single one, or split one `Person_Container` into two `Person_Container`'s.
- Now the fact that we use an array is a drawback (why?); a linked list would have been more practical in this case.
- So let's implement a container class called `Person_List` representing a list of `Persons`.

Convert Int_List Code to Person_List Code

- We don't want to implement all the list functionality from scratch. Suppose that, fortunately, we find a `Int_List` in our archives: a class that implements a doubly linked list of integers. We "reuse by copying" again: we copy the `Int_List`, rename the copy to `Person_List`, and replace all occurrences of `int` to `Person`.
- Oops ... we must be careful when we do the replacing; the `size()` member function of `Person_List` should still return an int ... .
- Of course, we still want to be able to print mailing labels. However, the function that we have doesn't work for a `Person_List`.
- Changing the type of the argument is trivial. A more serious problem is that our `Person_List` class has no method to retrieve a `Person` by specifying its index. That is typical for many implementations of a list.

Convert Int_List Code to Person_List Code ...

We could, of course, add a member function `get_person(i)` that retrieves a person by index, but what would that do to the running time of `print_mailing_labels()`?

Suppose that the original `Int_List`

- maintains a private pointer to the "current" element.
- `get_current( )` ⇒ get the current element.
- `set_first( )` ⇒ sets the pointer to the 1st item on the list.
- `set_next( )` ⇒ sets the pointer to the next element.
- `set_prev( )` ⇒ sets the pointer to the previous element.
- These functions return "-1" if there is nothing to point to.

Since we applied "reuse by copying", our `Person_List` class provides the same member functions as `Int_List` for manipulating the pointer to the "current" element in the list.

Example: print_mailing_labels() on Person_List

```
void print_mailing_labels(const Person_List& pl)
{
    if (pl.set_first() == -1)
        return;                                   // List is empty

    do
    {
        Person p = pl.get_current();
        cout << p.get_name() << endl;
        cout << p.get_address() << endl;
    } while (pl.set_next() != -1);      // End of list is reached
}
```

Example: invite_to_party() on Person_List


```
void invite_to_party(const Person_List& pl)
{
    if (pl.set_first() == -1)
        return;                                          // List is empty

    do
    {
        Person p = pl.get_current();
        string command = "cat party.txt | mail ";
        command += p.get_name();
        system( command.c_str() );     // Send invitation email
    } while (pl.set_next() != -1);     // End of list is reached
}
```

3 Concepts of Container Classes

In the previous examples, we can distinguish three concepts:

- the kind of container (list-based, array-based)
- the kind of objects stored in the container (`Person`, `int`)
- the kind of operations on the elements stored in the container ("do something for each element").

In our examples, there was a strong coupling between the three concepts:

- Whenever we change the type of the elements that we store, we have to re-implement the container (by copying and changing).
- Next, we also have to change the functions that deal with this container.

Similar Codes Again

Suppose that we want to search if a certain element is present in a container. Conceptually, the algorithm for searching an element is independent of the type of element: compare the elements in the container with the search element, until you have found it, or no such element is present. However, in our examples, we would need separate functions for

- searching a `Person` with a specific name in a `Person_Container`;
- searching a `Person` with a specific name in a `Person_List`;
- searching a specific value in a `Int_Container`;
- searching a specific value in a `Int_List`;

(We would also need to implement a `Int_Container` first; for instance, by copying and changing the `Person_Container`).

Generic Programming

- We see that strong coupling makes it impossible to reuse code without resorting to "code reuse by copying". This leads to programs that are very inflexible, and difficult to maintain and extend. Isn't there a better way?
- It is possible to remove (or strongly reduce) the strong coupling between containers, contained elements, and operations on the elements of the container by applying <u>Generic Programming</u>.
- GP means "**programming with types as parameters**".
- C++ supports GP through the <u>template</u> mechanism.

Example: max() by Copying

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

string max(const string& a, const string& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Example: max() by Generic Programming

Instead of copying, we write a single *template definition*.

```
template<typename T>
T max(const T& a, const T& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Example: Use of Template Function max()

Now we can use `max()` for any type of arguments, as long as the arguments can be compared by " >":

```
template<typename T>
T max(const T&a, const T&b)
{ if (a > b)
        return a;
  else return b;
}

main()
{ int x=4, y=8; string a("hulk"), b("jesse");
  cout << max(x,y) << " is the larger number\n";
  cout << max(a,b) << " is the stronger wrestler\n";
}
```

The Standard Template Library

- Containers are very common in programming, and several algorithms on container (searching for a specific element, searching for an element that satisfies some condition, sorting) occur in almost every non-trivial program.
- A lot of these general purpose containers can be found in the Standard Template Library, or STL for short.

To use the STL, we need an understanding of the following topics:

- templates
- containers
- iterators
- function objects
- operator overloading