CIS 330
C++ and Unix

Lecture 16

Templates

Containers

Containers are “holder” objects that store a collection of other objects

- Data structures like the stack, hash table, etc. are containers for holding data
- Instead of creating a large array of objects, create each one as needed using `new` and remove them using `delete`
  - Creating larger arrays also involves constructor and destructor overheads (even if some elements are not used)

Containers

Problem with data type can arise

- If you create a container of `int`, you can't use it for anything else
  - Using `void*` is not safe in C++
  - Copying and pasting new implementation is tedious and error prone

Templates

Templates implements the concept of **parameterized type**

- Type is a parameter
- Compiler only needs to know the exact data type when the actual code is generated from the template (at which point the "generic" data type is substituted)

Example

```
template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    if(index < 0 || index >= size)
      cout << "Index out of range" << endl;
      exit(-1);
  }
    return A[index];
};
```

```
int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
}
```

Example

```
template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};
```

```
template<class T>
T& Array<T>::operator[](int index) {
  if(index < 0 || index >= size){
      cout << "Index out of range" << endl;
      exit(-1)
  }
  return A[index];
}
```

```
int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++)
{
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
}
```

Header File

Normally, you only put declaration and inline functions in a header file

- This is because regular functions and variables allocate space and may cause link error (multiple definitions when included by multiple files)

However, with templates, this is OK

- Compiler does not allocate space for template code until the type is known (at instantiation e.g., `Array<int> ia;`)
- You almost always will see both the **declaration** and **definition** in the header file for templates
- Most compiler does have mechanism to put template definition in .cc files (but varies depending on the compiler)

Example

```
class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    if(top >= ssize)
      cout <<  "Too many
push()es") << endl;
    else
      stack[top++] = i;
  }
  int pop() {
    if(top <= 0)
      cout << "Too many pop()s"
<< endl;
    return stack[--top];
  }
};
```

```
int main() {
  IntStack is;
  // Push
  for(int i = 0; i < 20; i++)
    is.push(rand() % 1000);
  // Pop & print them:
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

Example

```
template<class T>
class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    if(top >= ssize)
     cout <<  "Too many push()es" << endl;
    stack[top++] = i;
  }
  T pop() {
    if(top <= 0)
      cout <<  "Too many pop()s" << endl;
    return stack[--top];
  }
  int size() { return top; }
};
```

Example

```
template<class T>
class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    if(top >= ssize)
     cout <<  "Too many push()es" << endl;
    stack[top++] = i;
  }
  T pop() {
    if(top <= 0)
      cout <<  "Too many pop()s" << endl;
    return stack[--top];
  }
  int size() { return top; }
};
```

Example

```
int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(rand() % 1000);
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;

  ifstream in("StackTemplateTest.cc");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```

Constants in Templates

```
template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }
  int length() const { return size; }
};
```

Constants in Templates

```
class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};
```

Constants in Templates

```
template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

```
int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Constants in Templates

```
template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

```
int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Constants in Templates

```
template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

```
template<class T, int size =
100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index
< size,
      "Index out of range");
    return array[index];
  }
  int length() const { return
size; }
};
```

```
return np->array[i];  // NOT ALLOWED - array is not public
```

Constants in Templates

```
template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

```
template<class T, int size =
100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index
< size,
      "Index out of range");
    return array[index];
  }
  int length() const { return
size; }
};
```

```
return np->[i] // NOT ALLOWED - invalid syntax
return np->operator[](i) // Explicitly calling the operator "function"
```

Constants in Templates

```
template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

```
template<class T, int size =
100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index
< size,
      "Index out of range");
    return array[index];
  }
  int length() const { return
size; }
};
```

```
return np->array[i];  // NOT ALLOWED - array is not public
return np[0][i]; // np[0] refers to the object pointed to by np. Now, you
can use [] to access the i-th element of Array.array.
```

Inheritance

```
template <class T>
class A {
    T i;
public:
    A() { i = 0; }
    A(T ii) : i(ii) {}
    void print() {
      cout << i << endl;
    }
};

template <class T>
class B : public A<T> {
    T j;
public:
    B() { j = 0; }
    B(T ii, T jj) :
      A<T>(ii), j(jj) {}
    void print();
};
```

```
template <class T>
void B<T>::print()
{
    A<T>::print();
    cout << j << endl;
}
int main()
{
    A<double> a(100.0);
    B<double> b(50.0,
200.0);

    a.print();
    b.print();

}
```

Inheritance

```
template <class T>
class A {
    T i;
public:
    A() { i = 0; }
    A(T ii) : i(ii) {}
    void print() {
      cout << i << endl;
    }
};

template <class T>
class B : public A<int> {
    T j;
public:
    B() { j = 0; }
    B(T ii, T jj) :
      A<int>(ii), j(jj) {}
    void print();
};
```

```
template <class T>
void B<T>::print()
{
    A<int>::print();
    cout << j << endl;
}
int main()
{
    A<double> a(100.0);
    B<double> b(50.0,
200.0);

    a.print();
    b.print();

}
```

Questions?

Upcasting

- A derived class is **a type** of the base class

```
enum note {C, Csharp, Cflat};

class Instrument {
public:
    void play(note) const {}
};

class Wind : public Instrument {};

void tune(Instrument &i) {
    i.play(C);
}
void testSeven()
{
    Wind flute;
    tune(flute);
}
```

Upcasting

```
enum note {C, Csharp, Cflat};

class Instrument {
public:
    void play(note) const { cout << "Instrument" << endl; }

};

class Wind : public Instrument {
public:
    void play(note) const { cout << "Wind" << endl; }
};

void tune(Instrument &i) {
    i.play(C);
}
```

```
int main()
{
    Wind w;
    Instrument* ip = &w; // upcast
    Instrument& ir = w;  // upcast
    tune(*ip);
    return 0;
}
```

Upcasting

```
enum note {C, Csharp, Cflat};
class Instrument {
public:
    virtual void  play(note) const { cout << "Instrument" << endl; }

};
class Wind : public Instrument {
public:
    void play(note) const { cout << "Wind" << endl; }
};
void tune(Instrument &i) {
    i.play(C);
}
```

```
int main()
{
    Wind w;
    Instrument *ip = &w; // upcast
    Instrument &ir = w; // upcast
    tune(*ip);
    return 0;
}
```

Upcasting and Copy-Constructor

```
class Parent {
private:
    int i;
public:
    Parent(int ii) : i(ii) { cout << "Parent(int ii)" << endl; }
    Parent(const Parent &b) : i(b.i) {
        cout << "Parent(const Parent &b)" << endl;
    }
    Parent() : i(0) { cout << "Parent()" << endl; }
    friend ostream& operator<< (ostream &os, const Parent &b) {
        return os << "Parent: " << b.i << endl;
    }
};
    Parent p(101);
    cout << p;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Parent: 101`

Upcasting and Copy-Constructor

```
class Member {
private:
    int i;
public:
    Member(int ii) : i(ii) { cout << "Member(int ii)" << endl; }
    Member(const Member &m) : i(m.i) {
        cout << "Member(const Member &m)" << endl;
    }
    friend ostream& operator<< (ostream &os, const Member &m) {
        return os << "Member: " << m.i << endl;
    }
};
    Member m(202);
    cout << m;
```

Upcasting and Copy-Constructor

- `Member(int ii)`
- `Member: 202`

Upcasting and Copy-Constructor

```
class Child : public Parent {
private:
    int i;
    Member m;
public:
    Child(int ii) : Parent(ii), i(ii), m(ii) {
        cout << "Child(int ii)" << endl;
    }
    friend ostream& operator<< (ostream &os, const Child &c) {
        return os << (Parent&) c << c.m << "Child: " << c.i << endl;
    }
};
    Child c(303);
    cout << c;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent: 303`
- `Member: 303`
- `Child: 303`

Upcasting and Copy-Constructor

- `Child c2(2);`
- `Child c3 = c2; // Child has no copy-constructor`
- `cout << c3;`

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent(const Parent &b)`
- `Member(const Member &m)`
- `Parent: 2`
- `Member: 2`
- `Child: 2`

Upcasting and Copy-Constructor

- Since Child has no explicitly defined copy-constructor, the compiler will **synthesize** one by calling the Parent copy-constructor and the Member copy-constructor

Upcasting and Copy-Constructor

- What if you make a mistake and leave out the base class' constructor
  - `Child(const Child& c) : i(c.i), m(c.m) {}`
- What would happen?
  - The default constructor will automatically be called for the base-class part of the Child (i.e., Parent), since that's what the compiler falls back on when it has no other choice of constructor to call

Upcasting and Copy-Constructor

```
class Parent {
private:
    int i;
public:
    Parent(int ii) : i(ii) { cout << "Parent(int ii)" << endl; }
    Parent(const Parent &b) : i(b.i) {
        cout << "Parent(const Parent &b)" << endl;
    }
    Parent() : i(0) { cout << "Parent()" << endl; }
    friend ostream& operator<< (ostream &os, const Parent &b) {
        return os << "Parent: " << b.i << endl;
    }
};
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent(const Parent &b)`
- `Member(const Member &m)`
- `Parent: 2`
- `Member: 2`
- `Child: 2`

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent()`
- `Member(const Member &m)`
- `Parent: 0`
- `Member: 2`
- `Child: 2`

Quiz

Question 1

Which of the following statements are NOT TRUE

1) Templates implements the concept of parameterized type, where the data type is also an **input parameter**
2) In inheritance, the derived class has all of the base class' members in its memory space, even the base class' **private** members.
3) Using the `delete` operator on a variable of type `void*` will cause a **runtime** error
4) When an array is created using the `new` operator (e.g., `new int[1000];`), you can only call the **default** constructor

Quiz

Question 2

Which of the following statements are NOT TRUE

1) Namespace definition can only appear at global scope, or nested within another namespace
2) You can alias namespace `Foo` using another name (e.g., `Bar`) using the following syntax:

   ```
   namespace Bar = Foo;
   ```

3) You have to include an **entire** namespace at once (i.e., you cannot specify/include just one member of a namespace)
4) **No** terminating semicolon is necessary when defining a namespace

Quiz

Question 3

Given the following piece of code, what will be the output?

```
class Integer {
    int i;
public:
    Integer(int ii) : i(ii) {}
    const Integer operator+(Integer& rv) const {
        Integer tmpInt(i + rv.i++);
        return tmpInt;
    }
    void print() { cout << i << endl; }
};

int main()
{
   Integer a(10);
   Integer b(20);
   Integer c = a + b;
   a.print();
   b.print();
   c.print();
   return 0;
}
```

1) 10
20
10

2) 10
21
30

3) 10
20
30

4) 10
30
30