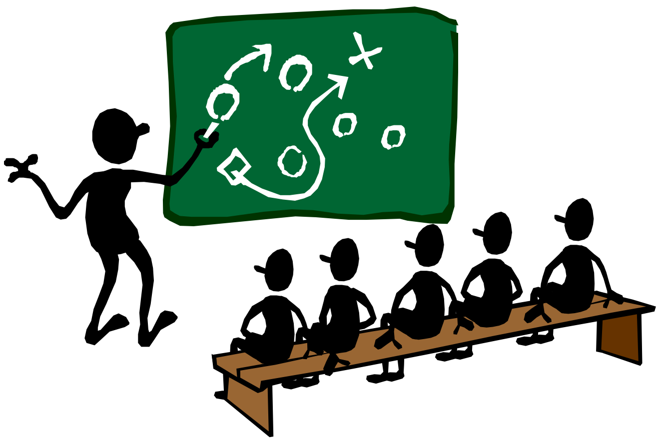
C++ Programming Language
Chapter 16 Templates



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

May 2011

Learning Objectives

- Function templates
  - syntax, definition, and usage
- Class templates
  - syntax, definition, and usage
- Templates and inheritance

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Generic Programming and Templates

- Object-oriented programming (OOP)
  - class, inheritance, (runtime) polymorphism
- Chapter 16 and 19 talk about generic programming
  - generic algorithms
  - containers
- Foundation of OOP ➔ Class
- Foundation of generic programming ➔ Template

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Motivation: Generic Algorithms

- Swap two integers

```
void swap(int& var1, int& var2) {
        int temp(var1);  var1 = var2;  var2 = temp; }
```

- Swap two characters

```
void swap(char& var1, char& var2) {
        char temp(var1);  var1 = var2;  var2 = temp; }
```

- How about swapping two doubles/strings/complex' …?
  - they basically do the same operation (swap)
  - only difference among them is type of object
- Generic algorithm
  - an algorithm can be expressed independently of representation details
- How do we achieve that in C++?

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Function Templates

- Function template definition
  - types can also be **PARAMETERS**!

```
template <typename T>
void swap(T& var1, T& var2) {  // need copy ctor and assignment operator
    T temp(var1); var1 = var2; var2 = temp; }

void f() {
    int i1 = 10, i2 = 100;
    char c1 = 'O', c2 = 'X';
    string s1("abc"), s2("xyz");
    swap(i1, i2);           // call swap<int>(int&, int&)
    swap(c1, c2);           // call swap<char>(char&, char&);
    swap(s1, s2);           // call swap<string>(string&, string&);
}
```

- Compiler is responsible to **instantiate** above 3 different swap functions

Function Template Definition (1/2)

- Template prefix

  e.g., ➔ template<typename T> or template<class T>

  – what inside **< >** is called template parameter list

  – template <typename T> and template<class T> can be used interchangeably (I personally prefer typename to class)

  – within the definition of template function, **T serves as a type name**

  – check the example in the previous slide

- Type arguments can be built-in or user-defined types

  – again, check the example in the previous slide

Function Template Definition (2/2)

- There can be more than one template parameter

  e.g., ➔ template<typename T1, typename T2>

- Template parameters can even be **non-type**

```
template <typename T, int d2>   // this code is just for demo; hardly useful
void print_2D_array(T arr[][d2], int d1) {
    for(int i = 0; i < d1; ++i)
        for(int j = 0; j < d2; ++j)        // d2 serves as an integer constant
            cout << arr[d1][d2] << endl; }

void f() {
int a[10][20];
    print_2D_array(a, 10);
     // call print_2D_array<int, 20>(int arr[][20], 10);
}
```

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Templates

Template Function Instantiation

- template<typename T>swap(T&, T&) is actually a large collection of definitions!
  – a definition for each feasible type!
- Compiler can deduce type and non-type arguments for a function template from the arguments of the function call
  – check the example in the previous slide
  – so-called compile-time polymorphism
- Compiler only instantiates (i.e., generates) actual function definitions when required (i.e., instantiation-on-demand)
  – for the example on Page 4, only for int, char, and string
- Write one template function definition
  ➔ it works for all types that might be needed
  ➔ generic programming

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example: Generic Sorting Algorithm

```
template<typename T>
void sort(T arr[], int size) {          // insertion sort, correct but slow
    for(int i = 1; i < size; ++i) {
        int j = i;
        for( ; j > 0; --j)
            if(arr[j] < arr[j - 1]) swap(arr[j], arr[j -1]);
}

void f(int ia[], char ca[], double da[], X xa[], int size) {
    sort(ia, size);          // ok, call sort<int>(int [], int)
    sort(ca, size);          // ok, call sort<char>(char [], int)
    sort(da, size);          // ok, call sort<double>(double [], int)
    sort(xa, size);          // not sure, it depends on whether there is an
}                            // "(arr[j] < arr[j - 1])" can be evaluated to a bool
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Function Template Overloading

Advanced

- Like other functions, template functions can be overloaded

```
template<typename T> T sqrt(T);
template<typename T> complex<T> sqrt(complex<T>);
double sqrt(double);        // in <cmath>

void f(complex<double> z) {
    sqrt(2);        // sqrt<int>(int)
    sqrt(2.0);      // double sqrt(double)
    sqrt(z);        // sqrt<double>(complex<double>)
}
```

- General rules for resolving overloaded functions
  - prefer ordinary functions to template functions
  - consider only the most specialized template function

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Function Templates Are Not Almighty

- In fact, a function template usually imposes constraints for its use

```
template <typename T>
void swap(T& var1, T& var2) {
    T temp(var1); var1 = var2; var2 = temp; }
```

- The above function template only works for those types with appropriate copy ctors and copy assignment operators

```
void f() {
    int a[10], b[10];
    swap(a, b);   // error! no copy ctor for integer array
                  //         no copy assignment operator for integer array
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Templates (1/2)

- Yes, you can have class templates too!
- Recall class IntArr in Chapter 8 & 10
  - it can only handle int array

```
template<typename T>
class Array {
    T *head; int size;
public:
    Array(int sz) : size(sz) { head = new T[size]; }  // ctor
    Array(const Array&);                               // copy ctor
    ~Array() { delete [] head; }                       // dtor
    Array& operator=(const Array&);                    // assignment operator
    T& operator[](int idx);                            // [] operator
    // …
};
```

Class Templates (2/2)

```
template<typename T>
Array<T>::Array(const Array<T>&  rhs)  : size(rhs.size) {  // copy ctor
    head = new T[size];
    for(int i = size - 1; i >= 0; --i)  head[i] = rhs.head[i];  }

template<typename T>
Array<T>& Array<T>::operator=(const Array<T>& rhs) {  // operator=
    T* tmp = new T[size = rhs.size];
     for(int i = size - 1; i >= 0; --i)  tmp[i] = rhs.head[i];
    delete[] head; head = tmp; return *this;  }

template <typename T>
T& Array<T>::operator[](int idx) {                    // [] operator
    if(idx >= 0 && idx < size) return head[idx];
    cerr << "Out-of-Range Error!\n";  exit(1);
};
```

A simple container example

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Templates

Template Class Instantiation

```
void f() {
    Array<int> iarr(100);      // explicitly specify template argument
                               // complete class(type) name: Array<int>
                               // i.e., iarr is an object of type Array<int>
    Array<char> carr(100);     // explicitly specify template argument
    iarr[20] = 101;
    carr[ iarr[20] ] = 'X';    // cause a runtime range-checking error!
}
```

- Again, compiler is responsible to generate all necessary code when required
- Similarly, a class template can
  - have more than one template parameter
  - have non-type template parameter
  - have default template parameter

Default Template Parameters (1/2)

```
template<typename T = int, int size = 100>
class Array {
    T head[size];
public:
    T& operator[](int idx);                             // [] operator
    // …
};

template <typename T, int size>
T& Array<T, size>::operator[](int idx) {               // [] operator
    if(idx >= 0 && idx < size) return head[idx];
    cerr << "Out-of-Range Error!\n";  exit(1);
};
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Default Template Parameters (2/2)

```
void f() {
    Array<int, 100> ia1;
    Array<> ia2;                  //  equivalently ➔ Array<int, 100>
    Array<char> ca;               //  equivalently ➔ Array<char, 100>
    ia1[20] = 101;
    Array<int, 100> ia3(ia1);     // use default copy ctor
    ia2 = ia3;                    // use default assignment operator;
    ca[ ia2[20] ] = 'X';          // cause a runtime range-checking error!
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Templates Are Not Almighty

- Similarly, a class template usually imposes constraints for its use
- For the previous
  template<typename T = int, int size = 100> class Array;

```
class X {
    int d;
public:
    X(int dd) : d(dd) { }    // no default ctor
};

void f() {
    Array<X, 100> xa;    // error! class X has no default ctor
}
```

Type Equivalence

```
void f() {
    Array<int, 100> a1;
    Array<long int, 100> a2;
    typedef long int Lint;
    Array<Lint, 100> a3;
    Array<Lint, 80> a4;
    Array<long int, 100 - 20> a5;
    // …
}
```

- a1 and a2 are of different types
- a2 and a3 are of same type
- a3 and a4 are of different types
- a4 and a5 are of same type

Friends and Templates

- A template class can have friends

```
template<typename T> class complex;         // forward declaration

template<typename T>
const complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs) {
    return complex<T>(lhs.real + rhs.real, lhs.image + rhs.image);  }

template<typename T>
class complex {
    T real, image;
public:
    complex(const T& re = T(), const T& im = T() ) : real(re), image(im) { }
    friend const complex operator+<>(const complex&, const complex&);
    /* … */  };

void f() {
    complex<int> a1, a2(2, 2), a3; a3 = a1 + a2;
    complex<double> b1, b2(2.0, 2.0), b3; b3 = b1 + b2;
}
```

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Templates and Inheritance

Advanced

- A template class can be derived from
  - non-template classes, or

    e.g., ➔ template<typename T> class D1 : public B1 { /* … */ };
  - template classes

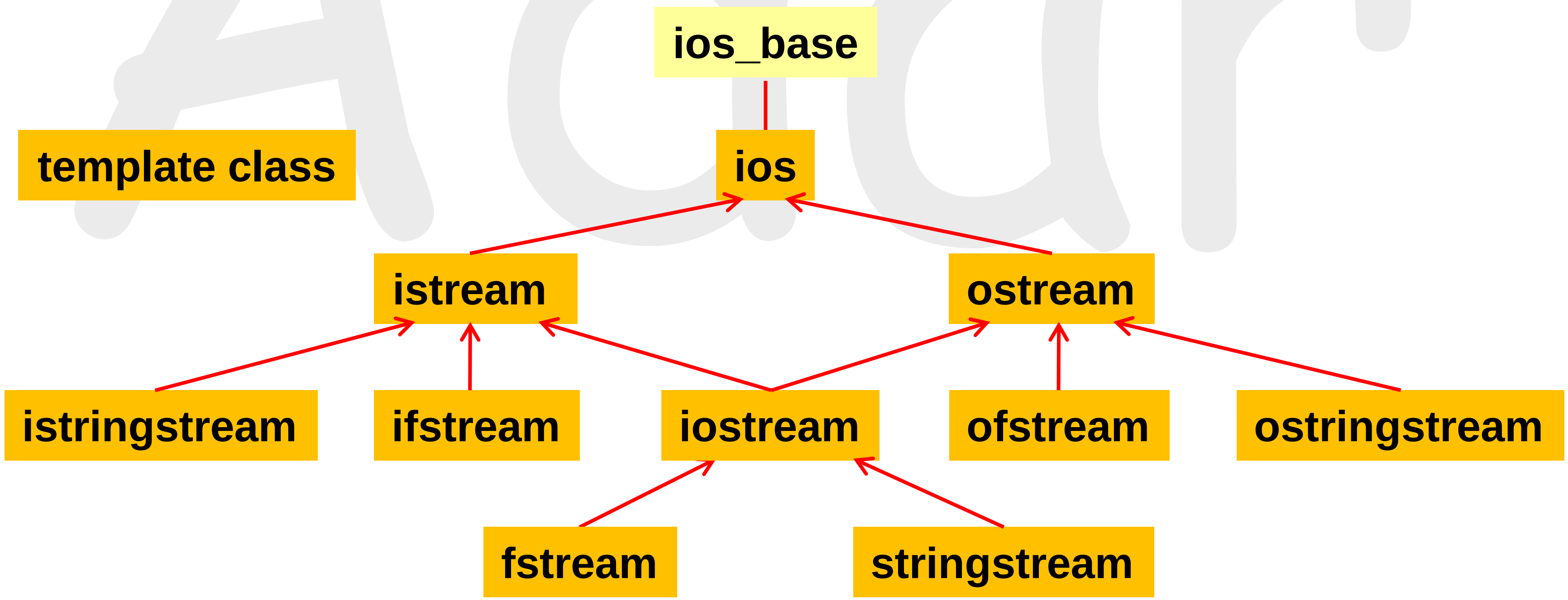
    e.g., ➔ template<typename T> class D2 : public B2<T> { /* … */ };
- A class instantiated (generated) from a class template is a perfectly ordinary class
  - even from a derived class template
  - follow the same set of rules (access controls, polymorphism, …)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Revisit Strings and Streams

- We've discussed strings and streams in Chapter 9 and 12
  - class string
  - class istream, class ostream, …
- Well, actually, they are all template classes!

```
typedef basic_string<char> string;
typedef basic_ios<char> ios;  typedef basic_istream<char> istream;
```



Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Template Development Strategy

- Develop ordinary functions/classes first
  - using actual data types
- Completely debug those ordinary functions/classes
- Then convert them to templates
  - replace type names with type parameter as needed
- Advantages
  - it's easier to solve non-template cases
  - deal with algorithms, not template syntax

Summary

- Templates ➔ type can be parameters
- Templates can enable
  - generic programming
  - containers (you will see in Chapter 19)
- Function templates
  - template function overloading
- Class templates
  - you can even define template classes derived from a template base class
- Default template parameters and non-type parameters
- basic_string and basic_ios (and its descendents) are template classes