

C++ and STL

Chan Siu Man
siuman@hkoi.org

Hong Kong Olympiad in Informatics

January 14, 2006



Agenda

1 C++ Specific Syntax

- Declaration of Variables
- Declaration of **struct**
- Pass by Reference

2 Object Oriented Programming

- Methods
- **new** and **delete** operators

3 Overloading

- Function Overloading

- operator Overloading

4 Template and Namespaces

- **templates**
- **namespaces**

5 Standard Template Library

- Design Pattern
- Iterators
- Containers
- Algorithms

6 Final Words on STL



Declaration of Variables

Variables need not be declared at the beginning of a block!
⇒ Variables can be declared only when needed!

```
#include <iostream>

using namespace std;

int main() {
    int n; /* declared at the beginning */
    cin >> n;

    int m; /* declared in the middle */
    cin >> m;

    cout << n << ' ' << m << endl;

    return 0;
}
```



Variables can be declared in `for` construct!¹

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;

    int a[20];
    for (int i = 0; i < n; ++ i)
        cin >> a[i];

    for (int i = 0; i < n; ++ i)
        cout << a[i] << endl;
    return 0;
}
```

Useful in avoiding using wrong index variables.

¹Different compilers may have slightly different behaviours. > <



Declaration of struct

You don't need `typedef` to name a `struct`.
Cleaner and more brief declaration.

```
struct Student {
    string lastName, firstName;
    int studentID, age;
}; /* <- don't forget this semi-colon */

struct Class {
    string name;
    vector<Student> students;
}; /* <- and also this one */
```



Pass by Reference

Changes made to a variable that is passed by reference is visible outside the subroutine.

```
int p1(int a, int &b, char &c) {
    int ret;

    /* some codes here */
    /* a is passed by value */
    ++ a;
    /* b and c are passed by reference */
    b = ++ c;

    return ret;
}
```



structs are usually passed by reference for speed up.

```
struct {
    string lastName, firstName;
    int age;
};

void older(Student &s) {
    ++ s.age;
}
```

To promise that a **struct** is not modified in a subroutine, use **const** keyword.

```
void print(const Student &s) {
    cout << s.lastName << ' ' << s.firstName << endl;
}
```



Methods

Subroutines, called methods, can be associated with objects, including **structs**.

```
struct Student {
    string lastName, firstName;
    int studentID;

    Student() {                /* constructor, optional */
        studentID = -1;
    }
    ~Student() {}             /* destructor, optional */

    void print() const { /* promise not to modify self */
        cout << lastName << ' ' << firstName << endl;
    }

    int getID() const { /* promise not to modify self */
        return studentID;
    }

    void setID(int id) { /* may modify self */
        studentID = id;
    }
};
```



Subroutines can also be defined out-of-line with scope resolution operator “::”.

```
struct Student {
    string lastName, firstName;
    int studentID;

    /* declarations */
    Student();
    ~Student();
    void print() const;
    int getID() const;
    void setID(int id);
};

/* definition */
Student::Student() {
    studentID = -1;
}

Student::~~Student() {
}

void Student::print() const {
    cout << lastName << ' ' << firstName << endl;
}

int Student::getID() const {
    return studentID;
}

void Student::setID(int id) {
    studentID = id;
}
```



new and delete operators

For dynamic memory management.
Constructors and destructors are called.

```
#include <iostream>

using namespace std;

int main() {
    int n;

    cin >> n;

    int *a;
    a = new int[n];

    for (int i = 0; i < n; ++ i)
        cin >> a[i];

    for (int i = 0; i < n; ++ i)
        cout << a[i] << endl;

    delete [] a;

    return 0;
}
```



Function Overloading

Same function name can be used for different subroutines, depending on the number, types and order of parameters passed. Default arguments can be specified.

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

long max(long a, long b) {
    if (a > b) return a;
    return b;
}

int max(const Student &s) {
    return s.age;
}

int max(int a) {
    return a;
}
```

They all refer to different subroutines with the same name.



operator Overloading

Let compilers understand how to compare and operate on objects, including **structs**.

```
struct Student {
    string lastName, firstName;
    int age;

    bool operator==(const Student &s) const {
        return lastName == s.lastName && firstName == s.firstName;
    }

    bool operator<(const Student &s) const {
        if (lastName != s.lastName) return lastName < s.lastName;
        return firstName < s.firstName;
    }

    bool operator>(const Student &s) const {
        return age > s.age;
    }

    bool operator==(int a) const {
        return age == a;
    }
};
```

Different operators can have completely different meanings!



operators are just some specially named subroutines associated with an object.

$$s1 == s2 \Leftrightarrow s1.operator==(s2)$$

List of operators that can be overloaded.

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	new	delete		

Arity, precedence and associativity of operators are inherited and cannot be redefined.

$$s1 + s2 * s3 \Leftrightarrow s1.operator+(s2.operator*(s3))$$

$$++ x \Leftrightarrow x.operator++()$$

$$x ++ \Leftrightarrow x.operator++(int)$$


templates

Generic **templates** enhance modularity and allow reusing of object codes by **class** substitution.

```
#include <iostream>
#include <string>

using namespace std;

template <class T>
T minimum(T a, T b) {
    if (a < b)
        return a;
    return b;
}

int main() {
    int i1 = 123, i2 = 321;
    double d1 = 2.718, d2 = 3.142;
    string s1 = "OI", s2 = "HK";

    cout << minimum(i1, i2) << endl;
    cout << minimum(d1, d2) << endl;
    cout << minimum(s1, s2) << endl;

    return 0;
}
```



namespaces

Collection of names (identifiers) to resolve conflict. Address a namespace with scope resolution operator “::”.

```
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Or with using keyword.

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```



Design Pattern

- Consider bubble sorting an array of integers and bubble sorting a linked list of integers. What is in common? What is the difference?
- Design Pattern allows the reuse of “solution patterns” to similar class of problems, even more abstract than the reuse of object codes.
- STL provides templates mainly of iterators, container classes and algorithms.



Iterators

- Loosely speaking, iterators are something that can be *dereferenced* and iterated by *increment*, *decrement* or *random access*.
- Iterators are abstraction of pointers. Pointers are examples of iterators.
- Useful iterators include
 - Forward Iterator;
 - Bidirectional Iterator; and
 - Random Access Iterator.
- The nature of the iterator is dependent on the properties of the associated container.



Containers

- A container is a class that contains other classes neatly.
- Containers are generalization of arrays. Arrays are examples of containers.
- A container usually has an associated iterator type that can be used to iterate through the container's elements. Use `container::iterator` to denote the associated iterator.
- Useful containers include:
 - `vector<T, Alloc>` (arrays);
 - `pair<T1, T2>` (pairs);
 - `queue<T, Sequence>` (queues);
 - `priority_queue<T, Sequence, Compare>` (heaps);
 - `set<Key, Compare, Alloc>` (RB-trees); and
 - `map<Key, Data, Compare, Alloc>` (RB-trees).



Algorithms

- Algorithms are template functions that operate on data, probably stored in containers and accessed by iterators.
- Useful algorithms include:

- `template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last, const
EqualityComparable& value);`
- `template <class InputIterator, class EqualityComparable>
iterator_traits<InputIterator>::difference_type count(InputIterator
first, InputIterator last, const EqualityComparable& value);`
- `template <class Assignable>
void swap(Assignable& a, Assignable& b);`
- `template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);`
- `template <class ForwardIterator>
inline ForwardIterator rotate(ForwardIterator first, ForwardIterator
middle, ForwardIterator last);`



- Other useful algorithms include:

- `template <class T> const T& min(const T& a, const T& b);`
- `template <class T> const T& max(const T& a, const T& b);`
- `template <class ForwardIterator>`
`ForwardIterator min_element(ForwardIterator first, ForwardIterator last);`
- `template <class ForwardIterator>`
`ForwardIterator max_element(ForwardIterator first, ForwardIterator last);`
- `template <class ForwardIterator, class LessThanComparable>`
`ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const LessThanComparable& value);`
- `template <class ForwardIterator, class LessThanComparable>`
`ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const LessThanComparable& value);`
- `template <class InputIterator, class T>`
`T accumulate(InputIterator first, InputIterator last, T init);`
- `template <class RandomAccessIterator>`
`void sort(RandomAccessIterator first, RandomAccessIterator last);`
- `template <class RandomAccessIterator>`
`void stable_sort(RandomAccessIterator first, RandomAccessIterator last);`



Final Words on STL

- Don't be scared by tens of lines of compilation errors when compiling programs with templates. Most of the time a single minor error accounts for all of the errors generated.
- Always look for the first cause of compilation errors.
- When nesting templates, don't forget to insert spaces between successive pointed brackets. (no ">>")
- Add **const** keyword to parameters and methods that are not modified or modifying. Proper use of **const** is essential for successful compilation.

