# Module03-13
# C++ 泛型编程：模板

# C++ 编程语言 - 泛型编程



- 楔子
- 语言基础
- 面向对象编程
- → 泛型编程

# 泛型编程

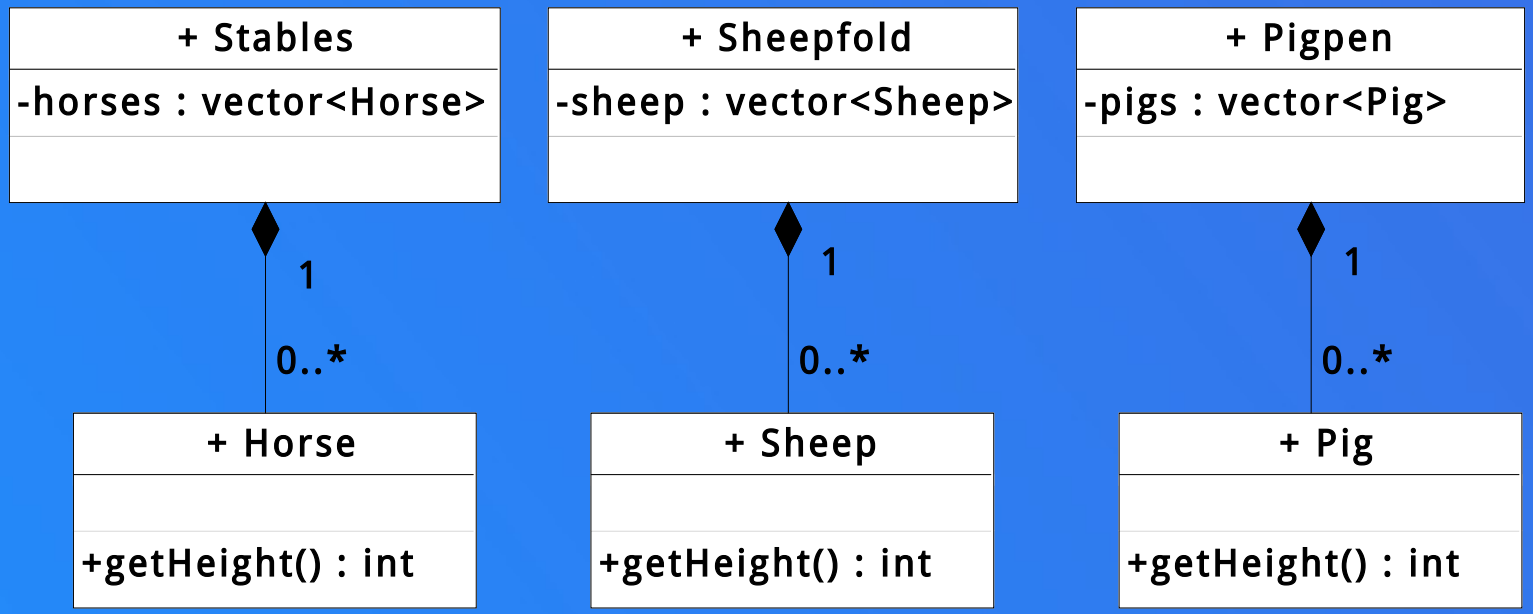


- 泛型编程 (Generic Programming)
  - 关于模板 (Templates)
  - 类模板 (Template Classes)
  - 函数模板 (Function Templates)
  - 用模板参数指定策略
  - 模板的特化与部分特化 (Specialization & Partial Specialization)
  - 派生与模板 (Derivation & Templates)

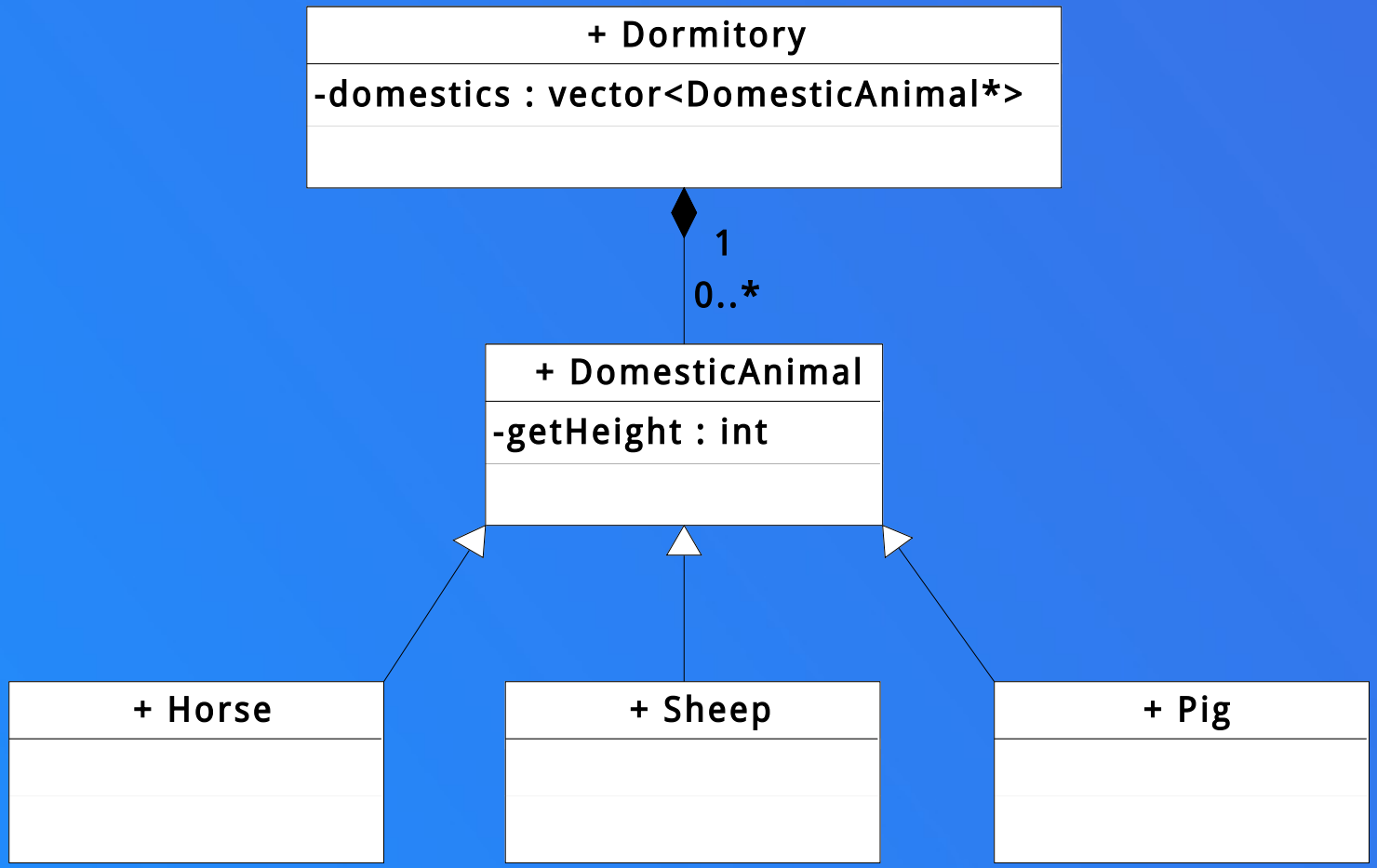
# 泛型编程 - 关于模板



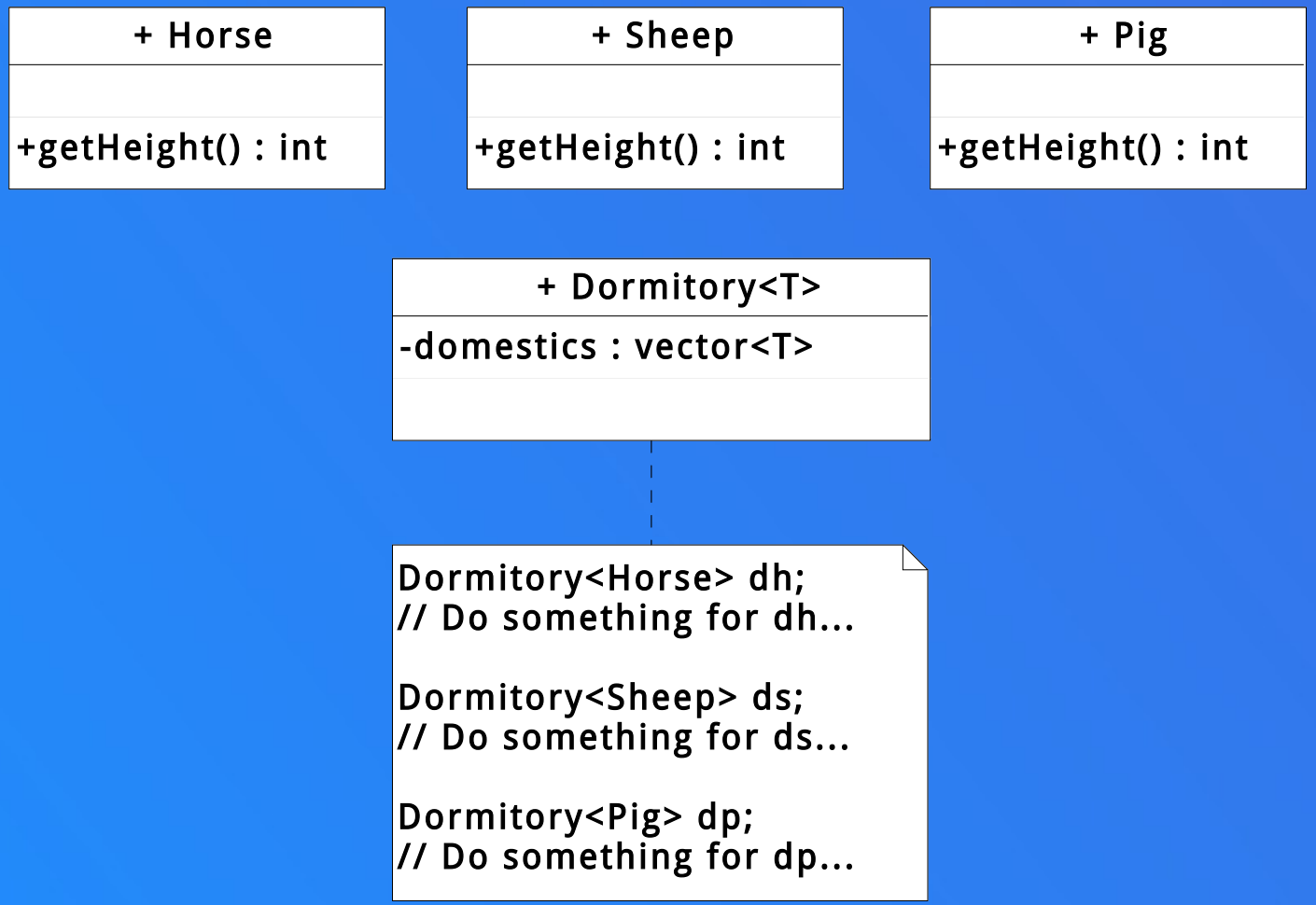
- 牲口圈舍的设计
  - 方案一



- 牲口圈舍的设计（续）
  - 方案二



- 牲口圈舍的设计（续）
  - 方案三



# 泛型编程 - 关于模板



- 关于模板的一些说法
  - 产生代码的代码
  - 实现编译期多态

- 一个 Array Class
  - 代码：

```
class Array {
    int _size;
    int _capacity;
    int* items;
public:
    typedef int* iterator;
    typedef const int* const_iterator;

    enum { DEFAUL_CAP = 16 };
    enum { RESERVED_SIZE = 5 };

    explicit Array(const int& capcity = DEFAUL_CAP);
    Array(const Array& other);
    Array& operator=(const Array& other);
    ~Array();

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
```

- 一个 Array Class（续）
  - 代码（续）：

```
    const int& operator[](const int& index) const;
    int& operator[](const int& index);

    void insert(const int& index, const int& value);
    void remove(const int& index);
    void append(const int& value);

    int size() const;
    int capacity() const;
    bool empty() const;

};
```

- 一个 Array Class（续）
  - Array 类的评议
    - Array 类实现了动态数组的功能，可以增加、删除、修改、获取数组中的元素，同时维护数组的 size
    - 不足之处：目前的 Array 只能容纳 int 型的对象，如果我们想用来容纳 double 、 Employee 等类型的对象，就力有不逮。
    - 分析：如上述需求，我们所需的 Array 在所有操作上完全一致，差别仅在于数组元素的类型不同，所以可以推广原先 Array 定义

# 泛型编程 - 类模板



- 将 Array 类参数化

```
template<typename T> // 注意，加了这行
class Array {
    int _size;
    int _capacity;
    T* items;    // 注意，原先是int(以下出现T的地方同此)
public:
    typedef T* iterator;
    typedef const T* const_iterator;

    enum { DEFAUL_CAP = 16 };
    enum { RESERVED_SIZE = 5 };

    explicit Array(const int& capcity = DEFAUL_CAP);
    Array(const Array& other);
    Array& operator=(const Array& other);
    ~Array();

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
```

# 泛型编程 - 类模板



- 将 Array 类参数化

```
    const T& operator[](const int& index) const;
    T& operator[](const int& index);

    void insert(const int& index, const T& value);
    void remove(const int& index);
    void append(const T& value);

    int size() const;
    int capacity() const;
    bool empty() const;

};
```

# 泛型编程 - 类模板



- 参数化后的 Array class 的使用

```
template<typename T>
void print(const Array<T>& a) {
    typename Array<T>::const_iterator it = a.begin();
    for (; it != a.end(); ++it)
        cout << *it << ' ';
    cout << endl;
}

int main() {
    Array<int> a(12);
    a.append(12);
    // ...
    a.append(199);
    a.insert(2, 289);
    a.remove(4);
    print(a);
    cout << a.capacity() << endl;
}
```

注意：在这里，编译器帮我们做了什么？

# 泛型编程 - 类模板



- 定义模板
  - 通常可以先定义一个具体的类型，在经过完整的测试后，再将其参数化，推广成 templates
  - 在类 body 外定义成员函数
    - 同普通的 class 函数一样，需要带 class 域，注意类的类型：

```
// 模板的实现
template<typename T>
Array<T>::Array(const int& capcity) :
    _size(0), _capacity(capcity) {
    items = new T[_capacity];
}
```

```
// 非模板实现(原函数)
Array::Array(const int& capcity) :
    _size(0), _capacity(capcity) {
    items = new int[_capacity];
}
```

# 泛型编程 - 类模板



- 模板实例化
  - 从一个模板类和一个模板参数生成一个类声明的过程，或者从一个模板函数和一个模板参数生成一个函数声明的过程，称为模板实例化
  - 模板实例化由编译器完成

```
// 参数为int 实例化成 class Array<int>
Array<int> a(12);

// 参数为double 实例化成 class Array<double>
Array<double> da(15);

// 参数为Employee 实例化成 class Array<Employee>
Array<Employee> ea(15);
```

# 泛型编程 - 类模板



- 模板参数
  - 模板参数可以是普通（实际）类型如 int，也可以是模板参数（如 T 等非实际类型），模板可以拥有一个或多个参数，如：

```
// capacity 在使用期间不能被更改
template<typename T, int capacity = 16>
class FixedArray { /* ... */ };
```

  - 一个模板参数甚至可以用作随后的模板参数的类型，如：

```
template<typename T, T temp>
class Cmp { /* ... */ };
```

  - 如果使用整型的模板参数，要求参数必须是 const 表达式，如：

```
void f(int k) {
    FixedArray<Employee, k> fa; // Error, k的值不是常量表达式
}
```

# 泛型编程 - 类模板



- 类型等价
  - 如下例中：
    - ea 、 da 虽然都是 Array ，但由于模板参数不同，所以属于 2 种不同的类型
    - fa1 、 fa2 、 fa3 则属于相同的类型

```
Array<Employee> ea;
Array<double> da;

FixedArray<double, 4 * 2> fa1;
FixedArray<double, 8> fa2;
typedef double dfloat;
FixedArray<dfloat, 8> fa3;
```

# 泛型编程 - 函数模板



- 一个简单的函数模板

```
template<typename T>
void mySwap(T& v1, T& v2) {
    T temp = v1;
    v1 = v2;
    v2 = tmp;
}

double d1 = 12.09, d2 = 18.02;
mySwap(d1, d2);  // same as: mySwap<double>(d1, d2);

int n1 = 120, n2 = 88;
mySwap(n1, n2); // same as: mySwap<int>(d1, d2);
```

- 函数模板的参数
  - 模板参数的推断
    - 编译器能够从一个调用推断出类型参数和非类型参数，但必须符合一个条件：由这个调用的函数参数列表能够唯一的标识出模板参数的一个集合
    - 注意：编译器不会对类模板的参数做任何推断
    - 如果自动推断失败，就需显式描述
    - 示例：

```
template<typename T>
T* getPtr() {
    T* ptr = new T();
    return ptr;
}

int main() {
    int* p = getPtr();  // Error
    int* q = getPtr<int>();  // OK
}
```

- 函数模板的重载
  - 函数模板重载解析规则与普通函数一致

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

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

  - 解析步骤：
    - 找出能参与这次重载解析的一族函数模板的特化。做这个的方式就是查看每一个函数模板，在假定没有其它同名函数模板或函数在作用域中的条件下，确定对它是否存在一组可以使用的模板参数，如果存在的话究竟是哪一组参数，如对于调用 sqrt(c)，将产生候选函数 sqrt<double>(complex<double>) 和 sqrt<complex<double>>(complex<double>)

- 函数模板的重载（续）
  - 解析步骤（续）：
    - 如果 2 个模板函数都可以调用，而其中一个比另一个更特化，在随后的步骤中就只考虑那个特化的模板函数，如 sqrt(c)，应该选择 sqrt<double>(complex<double>)，因为与 sqrt<T>(complex<T>) 匹配的调用也一定可以与 sqrt<T>(T) 匹配
    - 在这组函数上做重载解析，包括那些按常规函数考虑也应该加上去的常规函数。如果某个模板函数参数已经通过模板参数推断确定下来，这个参数就不能参与提升、标准转换、用户自定义转换。对于 sqrt(2)， sqrt<int>(int) 是确切匹配，优先于 sqrt(double)
    - 如果一个函数和一个特化具有同样精确的匹配，选函数，所以 sqrt(2.0) 选 sqrt(double)，不是 sqrt<double>(double)
    - 如果无法找到匹配，产生错误，如果最后得到了一个以上的匹配，该调用就产生二义性

- 函数模板的重载解析

```
template<typename T> T max(T, T);
const int s = 16;

void f() {
    max(1, 2);      // max<int>(1,2)
    max('a', 'b'); // max<char>('a', 'b')
    max(2.2, 5.9); // max<double>(2.2, 5.9)
    max(s, 8);      // max<int>(int(s), 8)

    max('a', 1);    // Error, 二义性: max<char>('a', 1)
                    // 还是 max<int>('a', 1) ?
    max(2.7, 4);    // Error, 二义性
}
```

  - 消除二义性
    - 通过显式描述
    - 通过增加声明

- 函数模板的重载解析（续）
  - 通过显式描述消除二义性

```
void f2() {
    max<int>('a', 1);    // max<int>(int('a'), 1)
    max<double>(2.7, 4);// max<double>(2.7, double(4))
}
```

  - 通过增加适当声明

```
inline int max(int i, int j) { return max<int>(i, j); }
inline double max(int i, double d) {
    return max<double>(i, d);
}
inline double max(double d, int i) {
    return max<double>(d, i);
}

inline double max(double d1, double d2) {
    return max<double>(d1, d2);
}
```

- 函数模板的重载解析（续）
  - 重载解析与类模板继承

```
template<typename T>
class B { /* ... */ };

template<typename T>
class D: public B<T> { /* ... */ };

template<typename T> void f(B<T>*);

void f2(B<int>* b, D<int>* d) {
    f(b); // f<int>(b)
    f(d); // f<int>(static_cast<int>*)(d), 标准转换 D<int>* ->
B<int>*
}
```

# 泛型编程 - 用模板参数指定策略



- 模板参数指定策略
  - 示例：标准库函数 sort()
    - 使用第二个模板参数（比较器）来指定比较的规则（如：降序或升序排列）

```
// STL sort()
template<typename Iterator, typename Compare>
inline void sort( Iterator first,  Iterator last, Compare
comp);

int main() {
    const int len = 5;
    int a[] = { 234, 23, 345, 98, 70 };
    sort(a, a + len, greater<int> ()); // 降序排列
    print(a, len);
    sort(a, a + len, less<int> ()); // 升序排列
    print(a, len);
}
```

# 泛型编程 - 用模板参数指定策略



- 默认模板参数
  - 与函数的默认参数类似，模板参数也可以指定默认参数

```
template<typename T, int size = 16>
class Array { /* ... */ };

template<typename T, typename Compare = Cmp<T> >
int compare(const Array<T>& a1, const Array<T>& a2);
```

  - 用于指定策略的模板参数又被称为“特征 (traits)”

# 泛型编程 - 模板特化



- 类模板的特化 (Specialization)
  - 例如：类模板 numeric_limits<>

```
template<typename T> struct numeric_limits; // 主类模板，通用情况

template<> struct numeric_limits<bool> ; // 适用于bool类型
template<> struct numeric_limits<char> ;// 适用于char类型
template<> struct numeric_limits<int> ; // 适用于int类型
template<> struct numeric_limits<double> ;// 适用于double类型

// 其它基本类型 ...
```

- 模板类成员函数的特化

```
template<>
void Array<const char*>::remove(const int& index) { /* ... */ }
```

# 泛型编程 - 模板特化



- 函数模板的特化

```
template<typename T>
bool greater(T a, T b) {
    return a > b;
}

// 针对 const char* 做特化
template<>
bool greater<const char*>(const char* a, const char* b) {
//bool greater(const char* a, const char* b) { // 简写方式
    return strcmp(a, b) > 0;
}
```

# 泛型编程 - 模板特化



- 部分特化 (Partial Specialization)

```
template<typename T>
class Array {
    int _size;
    int _capacity;
    T* items;
public:
    // ...
};

// 特化各种类型的指针
template<>
class Array<void*> {
    // ...
};

template<typename T>
class Array<T*>: private
Array<void*> {
    // ...
};
```

```
template<typename T>
class C { /* ... */ };

// 特化 const T
template<typename T>
class C<const T> { /* ... */ };

// ================================
template<typename T1, typename T2>
class X { /* ... */ };

// 特化一部分参数
template<typename T1>
class X<T1, int> { /* ... */ };

// ================================
template<int i, int j>
class Y { /* ... */ };

// 特化一部分参数
template<int i>
class Y<i, 6> { /* ... */ };
```

# 泛型编程 - 模板特化



- 部分特化 (Partial Specialization)
  - 部分特化的模板参数推导

```
Array<int*> a1; // <T*>是<int*>，即T是int
Array<int**> a2 // <T*>是<int**>，即T是int*
```

# 泛型编程 - 模板特化



- 关于特化的一些规则
  - 针对某个模板的特化并没有改变用户接口，只是手工生成一个模板的实例化声明
  - 一旦为某个模板做了特化，编译器将不会再为该特化所涉及的类型生成对应的实例化
  - 特化目的是为了解决通用模板不能精确解决的问题
  - 模板的特化版本依赖于通用模板，通用模板必须在所有特化模板之前声明（定义）
  - 通用模板和特化模板须在同一个名字空间
  - 模板特化的匹配的顺序：完全特化 -> 部分特化 -> 通用模板
  - 部分特化可以减少代码膨胀

# 泛型编程 - 派生与模板



- 派生与模板
  - 两者都是从已有的类型构造新类型的机制，通常用于编写利用各种共性的代码
  - 类继承利用虚函数实现运行期多态，而模板则是实现编译期多态（或称 参数式多态）
  - 如果对象间不需要某种层次的关系（编译期可以确定类型），最好将它们作为模板的参数；如果编译期无法确定这些类型，最好是将它们作为一个公共抽象类的派生类
  - 只要可能，为了达到性能上的要求，尽量使用模板

# 泛型编程 - 派生与模板



- 成员模板
  - 一个类（普通类）或类模板可以包含模板成员，如内部类模板、函数模板
  - 成员模板不能是 virtual 的

# 泛型编程 - 派生与模板



- 继承关系
  - 由同一个模板生成的 2 个类之间并不存在任何关系，即使是参数类型有继承关系：

```
class Mammal {};
class Lion: public Mammal {};

void f(Array<Mammal*>& ms) {
    // ...
    ms.append(new Lion());
    // ...
}

int main() {
    Array<Lion*> lions;
    f(lions); // Error
}
```

- 模板转换
  - 一个通用指针类型转换：

```
class Ptr {
    T* p;
public:
    Ptr(T* ptr);
    Ptr(const Ptr& ptr);
    template<typename T2> operator Ptr<T2>();
    //...
};

template<typename T>
template<typename T2> Ptr<T>::operator Ptr<T2>(){
    return Ptr<T2>(p);
}

void f(Ptr<Lion> pl) {
    Ptr<Mammal> pm = pl;  // OK
    Ptr<Lion> pl2 = pm;   // Error
}
```

# 泛型编程 - 使用 typename 标识类型



- typename 的使用
  - 在下面的函数模板的定义中，必须在 Array<T>::const_iterator 前加上 typename，否则该名字 const_iterator 将被当作是类 Array<T> 的一个 public 静态成员

```
template<typename T>
void print(const Array<T>& a) {
    typename Array<T>::const_iterator it = a.begin();
    for (; it != a.end(); ++it)
        cout << *it << ' ';
    cout << endl;
}
```

# 泛型编程 - Bjarne's Advices



- Bjarne's Advices
  - 需要多个参数类型的算法，使用模板
  - 用模板表述容器
  - 为指针的容器提供专门化，以减小代码规模
  - 总是在特化之前声明其通用模板
  - 定义你声明的每个特化
  - 用表述策略的对象进行参数化
  - 在转化为通用模板之前，在具体实例上排除错误
  - 如果运行时的效率非常重要，最好使用模板而不是派生类
  - 如果增加的各种变形而由不需重新编译很重要，使用派生类
  - 如果没有公共基类，使用模板而不是派生类