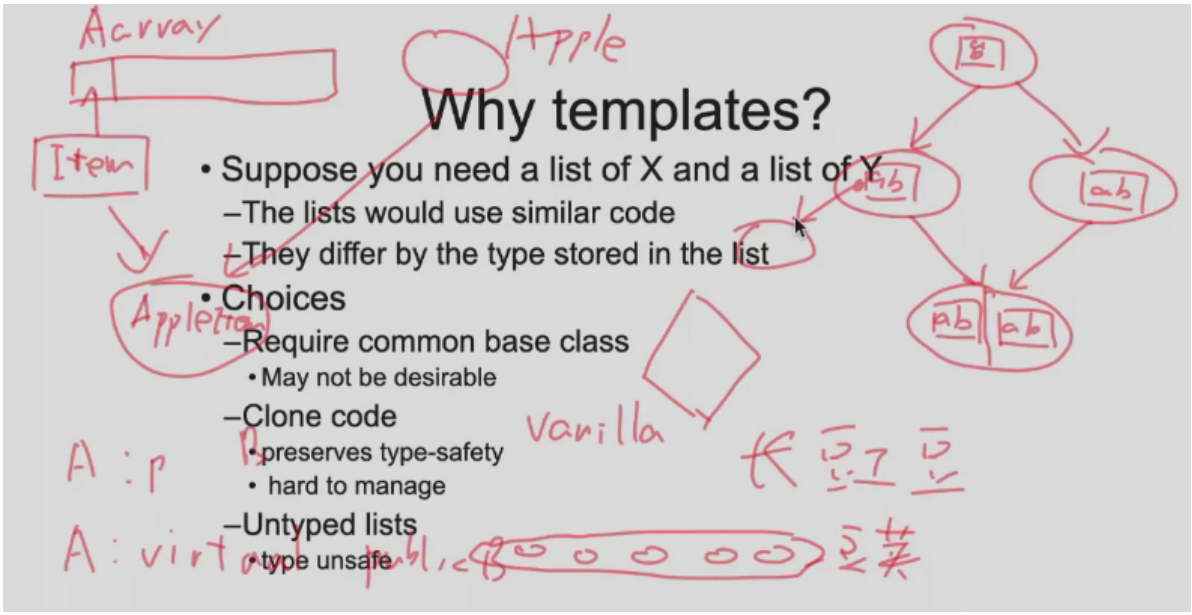
Templates



- 为了解决多继承问题中孙子可能会有两个父亲中带有的爷爷指针的区分问题，C++引入了**虚继承**的概念：
  - `A: virtual public B`：子类 `B` 中没有父类 `A`，父类 `A` 是在 `B` 的外面的；

---

1 Function Templates —— Generic Programming

The templates use **types parameters** in class or function definitions.

```
template < class T >        // meta
void swap( T& x, T& y ) {
    T temp = x;
    x = y;
    y = temp;
}
```

- The `template` keyword introduces the template;
- The `class T` specifies a parameterized type name;
  - `class` means any built-in type or user-defined type;
- Inside the template, use `T` as a type name:
  - types of arguments to the function, return type of the function, declare variables within the function.

1.1 Interactions

- **Only exact match** on types is used!!!
- **No conversion** operations are applied —— 参数不会做自动类型转换:

```
swap( int, int );       // ok
swap( double, double ); // ok
swap( int, double );    // error!!!
```

- even implicit conversions are ignored.

1.2 Overloading Rules

1. Check first for unique function match —— 第一步找有没有非模板函数的匹配;
2. Then check for unique function template match —— 第二步找有没有模板函数匹配;
3. Then do overloading on functions —— 最后尝试使用重载函数;
   - `eg.`

```
void mswap( float a, float b ) {
    float t = a;
    a = b;
    b = t;
    cout << "mswap(float)" << endl;
}
template < class T >
void mswap( T a, T b ) {
    T t = a;
    a = b;
    b = t;
    cout << "mswap(T)" << endl;
}
int main() {
    mswap( 1.0, 2.0 );  // 1.0, 2.0 are both double, so use template
    mswap( 1, 2 );      // 1, 2 are both int, so use template
    mswap( 1, 2.0 );    // conversions
}
-------------------------------------------------------------------------
mswap(T)
mswap(T)
mswap(float)
```

1.3 Function Instantiation

1. The compiler **deduces the template type** from the **actual arguments passed into** the function;
2. Can be **explicit**:
   - For example, if the parameter is not in the function signature(older compilers won't allow this)

```
template < class T >
void foo( void ) { /* ... */ }
foo<int>();
foo<double>();
```

2 Class Templates

- Classes parameterized by types
  - Abstract operations from the types being operated upon;
  - Define potentially infinite set of classes;
  - Another step towards reuse!
- Typical use: container classes:

```
stack <int> // a stack that is parameterized over int
list <Persion&>
queue <Job>
```

2.1 Define a Template Class

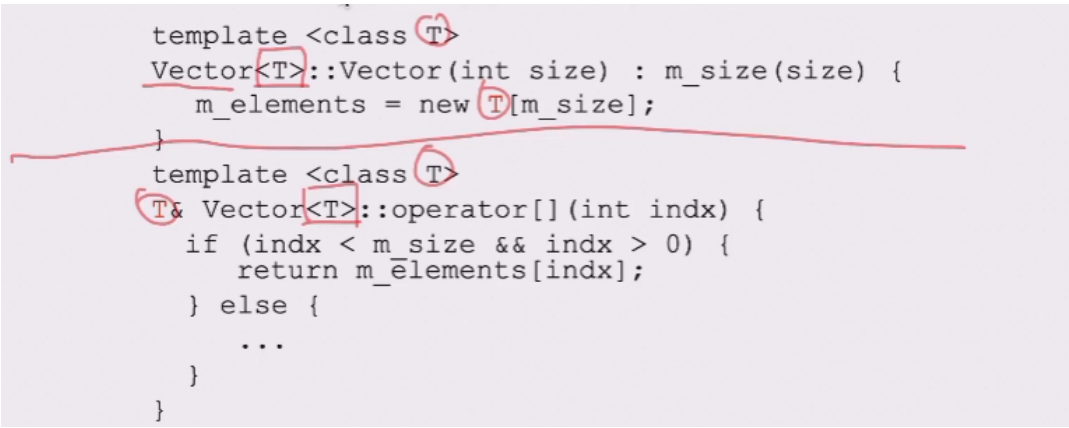
1. First we need to define the template class itself —— 首先把 `class` 的成员变量和成员函数定义出来：

```
template < class T >
class Vector {
public:
    Vector(int);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    T& operator[](int);
private:
    T* m_elements;
    int m_size;
};
```

2. Then we need to define the member function of the class —— 然后我们要定义 `class` 的成员函数：

```
template <class T>
Vector<T>::Vector(int size) : m_size(size) {
    m_elements = new T[m_size];
}
template <class T>
T& Vector<T>::operator[](int indx) {
    if (indx < m_size && indx > 0) {
        return m_elements[indx];
    } else {
        ...
    }
}
```

   - 要注意的是**所有的成员函数前面必须有** `template <class T>`，并且在函数原型中**必须要用** `Class<T>::` 来显式地指明这个类的变量类型：



2.2 Expression parameters

- Template arguments can be **constant expressions**.
- Non-Type parameters, can have a default argument:

```
template <class T, int bounds = 100>
class FixedVector {
public:
    FixedVector();
    // ...
    T& operator[](int);
private:
    T elements[bounds];     // fixed size array!
};
```

  - 这个做法可以让我们在**实例化一个模板类**的时候往我们类的源代码中**放入一个新的字面量**:

```
FixedVector<int, 50> v1;    // bounds = 50
FixedVector<int, 10*5> v2;  // bounds = 50
FixedVector<int> v3;        // uses default
```

3 Note: Templates is Declaration

Template class or template function are **both declaration** rather than definition, so if we put a template class in a `.h` file and its member function prototype, but the template member function definition in the `class.cpp` file, when we try to call the member function in `main.cpp`, there will be a **link error!！！**

```
/* Vector.h */
template <class T>
class Vector {
private:
    T* content;
    int size;
public:
    Vector( int s ):size(s) {}
    T& operator[]( int index );
};
-------------------------------------------------------------------------
--------;
```

```
/* Vector.cpp */
#include "Vector.h"
template <class T>
T& Vector<T>::operator[]( int index ) {
    return content[index];
}
--------------------------------------------------------------------------------;
/* main.cpp */
#include "Vector.h"
#include <iostream>
using namespace std;
int main() {
    Vector<int> v(100);
    v[0] = 10;
    cout << v[10] << endl;
}
--------------------------------------------------------------------------------;
C:\Users\Warriors\AppData\Local\Temp\ccUgdOe7.o:main.cpp:(.text+0x2b):
undefined reference to `Vector<int>::operator[](int)'
```

- 可以看到出现了链接时候报错，说找不到 `Vector<int>::operator[](int)` 这个函数的定义；

  - [x] **这是因为 template 的类以及类中的 template 函数其实都是和 inline function 是一样的：它们都是声明，所以是必须要放在 `.h` 文件中的，不能写在 `class.cpp` 文件里面的！！！**

- [ ] In general put the definition and the declaration for the template in the header file

4 Templates and Inheritance

- Templates can inherit from non-template classes:

```
template <class A>
class Derived : public Base { ... }
```

- Templates can inherit from template classes:

```
template <class A>
class Derived : public List<a> { ... }
```

- Non-template classes can inherit from tempates:

```
class SupervisorGroup : public List<Employee*> { ... }
```

Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune

- Review implementation
  - –Which types should be parameterized?
- Convert non-parameterized version into template
- Test against established test cases