

Programming Design

Templates, Vectors, and Exceptions

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Templates**
- The standard library `<vector>`
- Exception Handling

Warriors and wizards

- Recall our RPG game with warriors and wizards.
- Each character has a “unique” name, which is a **C++ string**.
 - An attribute used to distinguish members is called a **key**.
- It may be bad to use one’s name as its key.
- We may change it to an integer.
 - May we do better?

```
class Character
{
protected:
    static const int EXP_IV = 100;
    string name;
    int level;
    int exp;
    int power;
    int knowledge;
    int luck;
    // ...
};
```

Warriors and wizards

- If we change the type of name to be an integer, maybe one day we want to change it again.
 - The class still cannot be used for different key types.
- We may implement **two classes**, one for string and one for integer.
 - But the two classes will be almost identical. We are wasting our time.
 - Possible **inconsistency**.

```
class Character
{
protected:
    static const int EXP_IV = 100;
    string name;
    int level;
    int exp;
    int power;
    int knowledge;
    int luck;
    // ...
};
```

Templates

- We hope that our implementation is “general:”
 - For a character, we are flexible to choose the **type** of key.
- In C++, **templates** make this possible.
 - It can be applied on functions and classes.
 - It is not a feature of object-oriented programming.
- This is called **generic programming**.

Templates

- **C++ class templates** allows one to pass **a data-type argument** when:
 - Invoking a function defined with templates.
 - Creating an object whose class is defined with templates.
- In our example, objects of **Character** (and thus **Warrior** and **Wizard**) can be created with the actual key type passed as an argument.
 - `Warrior<string> w1("Alice", 10);`
 - `Wizard<int> w2(16, 5);`
- No need to write two implementations!

Template declaration

- To declare a type parameter, use the keywords **template** and **typename**.

```
template<typename T>
class TheClassName
{
    // T can be treated as a type inside the class definition block
};
```

- Some old codes write **class** instead of **typename**. Both are fine.

- We then do this to all member functions:

```
template<typename T>
T TheClassName<T>::f(T t)
{
    // t is a variable whose type is T
};
```

```
template<typename T>
void TheClassName<T>::f(int i)
{
    // follow the rule even if T is not used
};
```

Template invocation

- To instantiate an object, pass a type argument.

```
int main()
{
    TheClassName<int> a;
    TheClassName<double> b;
    TheClassName<AnotherClassName> c;
};
```

- The passed type will then replace all the **T**s in the class definition.

An example

- Let's start from an example with no class.
- When we invoke **f** with **f<double>**, the function is

```
void f(double t)
{
    cout << t;
}
```

- When we invoke **f** with **f<int>**, the function is

```
void f(int t)
{
    cout << t;
}
```

```
#include <iostream>
using namespace std;

template<typename T>
void f(T t)
{
    cout << t;
}

int main()
{
    f<double>(1.2); // 1.2
    f<int>(1.2); // 1

    return 0;
}
```

An example with two type parameters

- We may also have multiple type parameters.

```
#include <iostream>
using namespace std;

template<typename A, typename B>
void g(A a, B b)
{
    cout << a + b << endl;
}

int main()
{
    g<double, int>(1.2, 1.7); // 2.2
    return 0;
}
```

An example with classes

- The syntax of applying templates to classes is very similar.
 - Add the declaration line to the class definition.
 - Add the declaration line to all member function definitions.
 - For each member function definition, specify the type parameter.

```
int main()
{
    C<int> c;
    cout << c.f(10) << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;

template<typename T>
class C
{
public:
    T f(T i);
};

template<typename T>
T C<T>::f(T i)
{
    return i * 2;
}
```

Revising the classes

- Let's revise our definitions of **Character**, **Warrior**, **Wizard**, and **Team**.
- Let's start from **Character**.

```
template <typename KeyType>
class Character
{
protected:
    static const int EXP_LV = 100;
    KeyType name;
    int level;
    int exp;
    int power;
    int knowledge;
    int luck;
    void levelUp(int pInc, int kInc, int lInc);
public:
    Character(KeyType n, int lv, int po, int kn, int lu);
    virtual void beatMonster(int exp) = 0;
    virtual void print();
    KeyType getName();
};
```

Revising Character

```
template <typename KeyType>
Character<KeyType>::Character(KeyType n, int lv, int po, int kn, int lu)
    : name(n), level(lv), exp(pow(lv - 1, 2) * EXP_IV),
      power(po), knowledge(kn), luck(lu)
{
}

template <typename KeyType>
void Character<KeyType>::print()
{
    cout << this->name
         << ": Level " << this->level
         << " (" << this->exp << "/" << pow(this->level, 2) * EXP_IV << "), "
         << this->power << "-" << this->knowledge << "-" << this->luck << "\n";
}
```

Revising Character

```
template <typename KeyType>
void Character<KeyType>::levelUp(int pInc, int kInc, int lInc)
{
    this->level++;
    this->power += pInc;
    this->knowledge += kInc;
    this->luck += lInc;
}

template <typename KeyType>
KeyType Character<KeyType>::getName ()
{
    return this->name;
}
```

Revising Warrior

```
template <typename KeyType>
class Warrior : public Character<KeyType> // no class "Character"
{                                         // there is "Character<int>",
private:                                  // "Character<string>", etc.
    static const int PO_IV = 10;
    static const int KN_IV = 5;
    static const int LU_IV = 5;
public:
    Warrior(KeyType n, int lv = 0);
    void print();
    void beatMonster(int exp);
};
```

Revising Warrior

```
template <typename KeyType>
Warrior<KeyType>::Warrior(KeyType n, int lv)
    : Character<KeyType>(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}

template <typename KeyType>
void Warrior<KeyType>::print()
{
    cout << "Warrior ";
    Character<KeyType>::print();
}

template <typename KeyType>
void Warrior<KeyType>::beatMonster(int exp)
{
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * Character<KeyType>::EXP_LV) // Why?
        this->levelUp(PO_LV, KN_LV, LU_LV);
}
```


Revising Wizard

```
template <typename KeyType>
class Wizard : public Character<KeyType> // no class "Character"
{                                       // there is "Character<int>",
private:                                // "Character<string>", etc.
    static const int PO_LV = 10;
    static const int KN_LV = 5;
    static const int LU_LV = 5;
public:
    Wizard(KeyType n, int lv = 0);
    void print();
    void beatMonster(int exp);
};
```

Revising Wizard

```
template <typename KeyType>
Wizard<KeyType>:: Wizard(KeyType n, int lv)
    : Character<KeyType>(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}

template <typename KeyType>
void Wizard<KeyType>::print()
{
    cout << "Wizard ";
    Character<KeyType>::print();
}

template <typename KeyType>
void Wizard<KeyType>::beatMonster(int exp)
{
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * Character<KeyType>::EXP_LV)
        this->levelUp(PO_LV, KN_LV, LU_LV);
}
```

Revising Team

```
template <typename KeyType>
class Team
{
private:
    int memberCount;
    Character<KeyType>* member[10];
public:
    Team();
    ~Team();
    void addWarrior(KeyType name, int lv);
    void addWizard(KeyType name, int lv);
    void memberBeatMonster(KeyType name, int exp);
    void printMember(KeyType name);
};
```

Revising Team

```
template <typename KeyType>
Team<KeyType>::Team()
{
    this->memberCount = 0;
    for(int i = 0; i < 10; i++)
        member[i] = nullptr;
}

template <typename KeyType>
Team<KeyType>::~~Team()
{
    for(int i = 0; i < this->memberCount; i++)
        delete this->member[i];
}
```

Revising Team

```
template <typename KeyType>
void Team<KeyType>::addWarrior(KeyType name, int lv) {
    if(memberCount < 10) {
        member[memberCount] = new Warrior<KeyType>(name, lv);
        memberCount++;
    }
}

template <typename KeyType>
void Team<KeyType>::addWizard(KeyType name, int lv) {
    if(memberCount < 10) {
        member[memberCount] = new Wizard<KeyType>(name, lv);
        memberCount++;
    }
}
```

Revising Team

```
template <typename KeyType>
void Team<KeyType>::memberBeatMonster(KeyType name, int exp) {
    for(int i = 0; i < this->memberCount; i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->beatMonster(exp);
            break;
        }
    }
}
```

```
template <typename KeyType>
void Team<KeyType>::printMember(KeyType name) {
    for(int i = 0; i < this->memberCount; i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->print();
            break;
        }
    }
}
```

In the main function

```
int main()
{
    Team<string> t;

    t.addWarrior("Alice", 1);
    t.memberBeatMonster("Alice", 10000);
    t.addWizard("Bob", 2);
    t.printMember("Alice");

    Team<int> t2;

    t2.addWarrior(1, 1);
    t2.memberBeatMonster(1, 10000);
    t2.addWizard(2, 2);
    t2.printMember(1);

    return 0;
}
```

One final remark

- An **operation** may need a special definition for a given type.

```
template <typename KeyType>
void Team<KeyType>::memberBeatMonster(KeyType name, int exp) {
    for(int i = 0; i < this->memberCount; i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->beatMonster(exp);
            break;
        }
    }
}
```

- What if **KeyType** is a **class** rather than a basic data type?
- We then need to do **operator overloading**.

Outline

- Templates
- **The standard library `<vector>`**
- Exception Handling

A good reason to use templates

- For strings:
 - We use a character array to represent a C string.
 - We use the class `string` to represent a C++ string.
 - The latter is to **embed the former into a class** and **add useful functions**.
- How about integers?
- We may implement a class with an embedded dynamic integer array and useful functions.
 - How about fractional numbers?
 - How about any other type of items?
- All we need is a class with an embedded dynamic array **for something**.
 - Perfect for templates!

The standard library `<vector>`

- In fact, C++ provides such a template-based class.
- In C++, there is a **standard template library (STL)**.
 - It provides containers, iterators, algorithms, and functions.
- **The class `vector`** with templates is defined and implemented in the standard library `<vector>`.
- It is just a “**dynamic vector**” of any type.
 - It is a class with an embedded one-dimensional dynamic array.
 - It has many useful member functions (including overloaded operators).
 - It is implemented with templates.

The standard library <vector>

- A vector is very easy to use.
 - To create a vector, indicate the type of items:

```
vector<int> v1; // integer vector
vector<double> v2; // double vector
vector<Warrior> v3; // Warrior vector
```

- Member functions that modifies a vector:
 - **push_back()**, **pop_back()**, **insert()**, **erase()**, **swap()**, **=**, etc.
- Member functions for one to access a vector element:
 - **[], front()**, **back()**, etc.
- Member functions related to the capacity:
 - **size()**, **max_size()**, **resize()**, etc.

The standard library <vector>

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(vector<int> v)
{
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}
```

```
int main()
{
    vector<int> v;
    cout << v.size() << endl;
    cout << v.max_size() << endl;
    v.push_back(10);
    v.push_back(9);
    v.push_back(8);
    printVector(v); // 10 9 8
    v.pop_back();
    v.push_back(5);
    printVector(v); // 10 9 5

    return 0;
}
```

Rewriting Team using vector

- Recall our implementation of **Team**:
- Let's replace the static array by a vector.

```
template <typename KeyType>
class Team
{
private:
    int memberCount;
    Character<KeyType>* member[10];
public:
    Team();
    ~Team();
    void addWarrior(KeyType name, int lv);
    void addWizard(KeyType name, int lv);
    void memberBeatMonster(KeyType name, int exp);
    void printMember(KeyType name);
};
```

Rewriting Team using vector

- This vector will still store **Character***.
- Of course, it is now **Character<KeyType>***.

```
template <typename KeyType>
class Team
{
private:
    vector<Character<KeyType>*> member;
public:
    Team();
    ~Team();
    void addWarrior(KeyType name, int lv);
    void addWizard(KeyType name, int lv);
    void memberBeatMonster(KeyType name, int exp);
    void printMember(KeyType name);
};
```

Rewriting Team using vector

- Now (it seems that) we do not need the constructor and destructor.

```
template <typename KeyType> // old
Team<KeyType>::Team()
{
    this->memberCount = 0;
    for(int i = 0; i < 10; i++)
        member[i] = nullptr;
}

template <typename KeyType>
Team<KeyType>::~~Team()
{
    for(int i = 0; i < this->memberCount; i++)
        delete this->member[i];
}
```

```
template <typename KeyType> // new
Team<KeyType>::Team()
{
}

template <typename KeyType>
Team<KeyType>::~~Team()
{
}
```


Rewriting Team using vector

- To add a member, push a **pointer** of its address into the vector.

```
template <typename KeyType>
void Team<KeyType>::addWarrior(KeyType name, int lv)
{
    Warrior<KeyType>* wPtr = new Warrior<KeyType>(name, lv);
    this->member.push_back(wPtr);
}

template <typename KeyType>
void Team<KeyType>::addWizard(KeyType name, int lv)
{
    Wizard<KeyType>* wPtr = new Wizard<KeyType>(name, lv);
    this->member.push_back(wPtr);
}
```

Rewriting Team using vector

- Why using dynamic memory allocation? What is wrong below?

```
template <typename KeyType>
void Team<KeyType>::addWarrior(KeyType name, int lv)
{
    Warrior<KeyType> w(name, lv); // no DMA
    this->member.push_back(&w);
}

template <typename KeyType>
void Team<KeyType>::addWizard(KeyType name, int lv)
{
    Wizard<KeyType> w(name, lv); // no DMA
    this->member.push_back(&w);
}
```

Rewriting Team using vector

```
template <typename KeyType>
void Team<KeyType>::memberBeatMonster(KeyType name, int exp) {
    for(int i = 0; i < this->member.size(); i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->beatMonster(exp);
            break;
        }
    }
}

template <typename KeyType>
void Team<KeyType>::printMember(KeyType name) {
    for(int i = 0; i < this->member.size(); i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->print();
            break;
        }
    }
}
```

Remarks

- We change the **private data structure**, not those public interfaces.
 - No one needs to modify her code of using **Team**.
- Do we need a constructor?
 - No: The vector will be initially empty. All its initialization tasks will be done automatically.
- Do we need a destructor?

```
template <typename KeyType>
class Team
{
private:
    vector<Character<KeyType>*> member;
public:
    Team();
    ~Team();
    void addWarrior(KeyType name, int lv);
    void addWizard(KeyType name, int lv);
    void memberBeatMonster(KeyType name, int exp);
    void printMember(KeyType name);
};
```

Remarks

- We do need a **destructor** to release those dynamically created **Warrior** and **Wizard**.
 - Before we remove an element in the vector (which is a pointer), we should release the pointed space.
- Following the same idea, we now know how to implement a function to remove a team member.
 - Locate the member, release the space, and then remove that vector element.
- (Personal suggestion) **Try not to use vector** unless you are able to implement something with the same capability.

```
template <typename KeyType>
Team<KeyType>::~~Team()
{
    while (this->member.size() > 0)
    {
        delete this->member.back();
        this->member.pop_back();
    }
}
```

Outline

- Templates
- The standard library `<vector>`
- **Exception Handling**

Exceptions

- **Exceptions** are those thing that are not expected to happen.
 - When one writes a program, that typically refers to **logic** or **run-time** errors.
- Consider the following example:

```
#include <iostream>
using namespace std;

void f(int a[], int n)
{
    int i = 0;
    cin >> i;
    a[i] = 1; // run-time error?
}
```

```
int main()
{
    int a[5] = {0};
    f(a, 5);
    for(int i = 0; i < 5; i++)
        cout << a[i] << " ";
    return 0;
}
```

Exceptions

- Some **checks** can be helpful:

```
#include <iostream>
using namespace std;

bool f(int a[], int n)
{
    int i = 0;
    cin >> i;
    if(i < 0 || i > n)
        return false;
    a[i] = 1;
    return true;
}
```

```
int main()
{
    int a[5] = {0};
    f(a, 5);
    for(int i = 0; i < 5; i++)
        cout << a[i] << " ";
    return 0;
}
```

- The caller can check the value returned by **f()** to do appropriate responses.

Exceptions

- Some **checks** may not be enough.
 - If the function returns **false** due to **multiple reasons**, the client will not see the reasons.
 - It is hard to **send messages** to the client about the error (do not print out an error message on the screen!).
 - We **cannot enforce** the client to respond to the returned value.
- C++ (and many other modern languages) offers **exception handling**.
 - A mechanism for handling **logic** or **run-time error**.
 - A function can report the occurrence of an error by **throwing an exception**.
 - One **catches an exception** and then respond accordingly.

Try and catch

- In C++, we use a **try** block and **catch** blocks.
- A **try** block must be followed by **at least one catch** block.
- Ideally, we include only statements that may throw exceptions in a **try** block.
- Curly brackets are always needed!

```
try
{
    // statements that may throw exceptions
}
catch(ExceptionClass identifier) // this kind?
{
    // responses
}
catch(AnotherExceptionClass identifier) // that kind?
{
    // other responses
}
```

Try and catch

- When a statement (function or method) in a **try** block causes an exception:
 - Control **ignores the rest statements** in the **try** block.
 - Control passes to the catch block **corresponding to the exception** (if any).
 - After the catch block executes, control passes to statements after this try-catch block.
- If there is no applicable catch block for an exception, **abnormal program termination** usually occurs.
- If an exception occurs in the middle of a try block, the **destructors** of all (static) objects local to that block are called.
 - This is to ensure that all resources allocated in that block are released.
 - It is suggested **not to dynamically allocate** anything inside a try block.

Example: `string::replace()`

- The `replace()` function of C++ strings is easy to use.

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
    s.replace(i, 1, ".");
}
```

```
int main()
{
    string s = "12345";
    int i = 0;
    cin >> i;
    g(s, i);
    cout << s << endl;
    return 0;
}
```

- It is defined to be able to **throw** an **out_of_range exception**.
 - `out_of_range` is a class defined in `<stdexcept>`.
 - If we do not respond to the exception, the program terminates abnormally.

Example: `string::replace()`

- Let's try and catch!

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
    try {
        s.replace(i, 1, ".");
    }
    catch(out_of_range e) {
        cout << "...\\n";
    }
}
```

```
int main()
{
    string s = "12345";
    int i = 0;
    cin >> i;
    g(s, i);
    cout << s << endl;
    return 0;
}
```

Example: `string::replace()`

- We may also try and catch in the caller:

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
    s.replace(i, 1, ".");
}
```

- A thrown exception will be passed to callers until one catches it.
 - If no one catches it, the program terminates abnormally.

```
int main()
{
    string s = "12345";
    int i = 0;
    cin >> i;
    try {
        g(s, i);
    }
    catch(out_of_range e) {
        cout << "...\\n";
    }
    cout << s << endl;
    return 0;
}
```

Standard exception classes

- In the C++ standard library, we have the following standard exception classes:
 - Inheritance and polymorphism!

```
try {  
    g(s, i);  
}  
// this also works  
catch (logic_error e) {  
    cout << "... \n";  
}
```

- Include <stdexcept> to use them.

exception

logic_error

domain_error

invalid_argument

length_error

out_of_range

runtime_error

range_error

overflow_error

underflow_error

Throwing an exception

- We may also **throw an exception** by ourselves.

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(logic_error)
{
    int i = 0;
    cin >> i;
    if(i < 0 || i > n)
        throw logic_error("...");
    a[i] = 1;
}
```

```
int main()
{
    int a[5] = {0};
    f(a, 5);
    for(int i = 0; i < 5; i++)
        cout << a[i] << " ";
    return 0;
}
```

- This **enforce** the client to **catch** the exception (to avoid forced termination).

Throwing an exception

- Let the client **catch** the exception:

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(logic_error)
{
    int i = 0;
    cin >> i;
    if(i < 0 || i > n)
        throw logic_error("...");
    a[i] = 1;
}
```

```
int main()
{
    int a[5] = {0};
    try {
        f(a, 5);
    }
    catch(logic_error e) {
        cout << e.what();
    }
    for(int i = 0; i < 5; i++)
        cout << a[i] << " ";
    return 0;
}
```

- what ()** returns the message generated when throwing an exception.

Modifying the function header

- Functions that throw an exception may have a **throw clause** at the end of their headers.
 - This restricts the exceptions that a function can throw.
 - Omitting a **throw** clause allows a function to throw any exception.
- To allow multiple types of exceptions:

```
void f(int a[], int n) throw(type1, type2)
```

- The documentation of a function (or method) should indicate any exception it might throw.

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n)
    throw(logic_error)
{
    int i = 0;
    cin >> i;
    if(i < 0 || i > n)
        throw logic_error("...");
    a[i] = 1;
}
```

Functions that do not(?) throw exception

- If a function will **never** throw an exception, one may **indicate this explicitly**.
- For example, (in C++ 11) the function **length()** of the class **string** is actually defined as:

```
size_t length() const noexcept;
```

- This means that this function never throws an exception.
- When one calls a function, it is good to know that it may (or will never) throw an exception.
 - Therefore, indicate this if you know that is true.
- It is the programmer's responsibility make sure that the function indeed does not throw an exception; the compiler does not check anything.

Defining your own exception classes

- C++ Standard Library supplies a number of exception classes.
- You may also want to define **your own exception class**.
 - This helps your program communicate better to your clients.
 - Your own exception classes should **inherit from standard exception classes** for a standardized exception working interface.

```
#include <stdexcept>
using namespace std;

class MyException : public exception
{
public:
    MyException(const string& msg = "")
        : exception(msg.c_str()) {}
};
```

Defining your own exception classes

- Let's use our own exception class:

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(MyException)
{
    int i = 0;
    cin >> i;
    if(i < 0 || i > n)
        throw MyException("...");
    a[i] = 1;
}
```

```
int main()
{
    int a[5] = {0};
    try {
        f(a, 5);
    }
    catch(MyException e) {
        cout << e.what();
    }
    for(int i = 0; i < 5; i++)
        cout << a[i] << " ";
    return 0;
}
```

Applying these techniques

- Exception handling may be applied to **Team**.
- When the team size limit is reached, throw an exception.

```
template <typename KeyType>
void Team<KeyType>::addWarrior(KeyType name, int lv) throw (MyException) {
    if(memberCount < 10) {
        member[memberCount] = new Warrior<KeyType>(name, lv);
        memberCount++;
    }
    else
        throw MyException("...");
}
```

Applying these techniques

- When a non-existing member is searched for, throw an exception.

```
template <typename KeyType>
void Team<KeyType>::memberBeatMonster(KeyType name, int exp) throw (MyException) {
    bool isFound = false;
    for(int i = 0; i < this->memberCount; i++) {
        if(this->member[i]->getName() == name) {
            this->member[i]->beatMonster(exp);
            isFound = true;
            break;
        }
    }
    if(isFound == false)
        throw MyException("...");
}
```