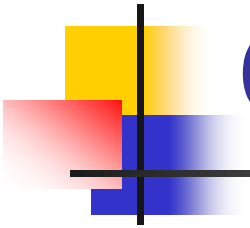
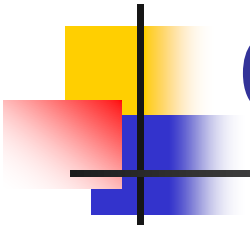
Week 4



Chapter 7: Templates

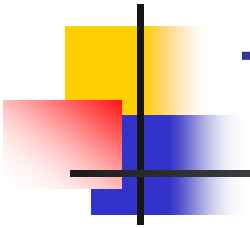
Outline:

- Motivation
- Function templates
- Class templates



Generic functions and classes

- Can we write a function or implement a class to handle as many types of data as possible yet they are written once only?
- Yes, we can, with C++ templates
- C++ Templates are functions or classes that are written for one or more types not yet specified
- The standard template library (STL) is full of solutions in templates to manage collections of data.



function min()

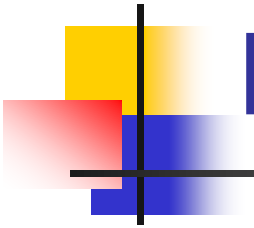
```
int min(int left, int right){   return left < right ? left : right;}
```

This function can be used to find the min of a pair of int only.

We have to write different versions of this function to find the min of a pair of floats, strings, …

```
int min(float left, float right){return left < right ? left : right;}
```

```
int min(string left, string right){return left < right ? left : right;}
```



Function Templates

- A function template is a function with data type as parameters which can be instantiated with particular types to generate a function definition
- Written once, and instantiated as many times as different data types are used

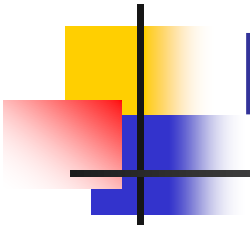
A Small Function Template Example (1)

```
$ cat getsize.cpp
#include <iostream>
using namespace std;
// getSize() is a function template
// T is a parametrized type
template <typename T>
int getSize(T & p) {
return sizeof(T);
}
```

A Small Function Template Example (2)

```
int main() {
        bool b; char c;
        short s; int i; long long l;
        float f; double d; long double g;
        cout << getSize(b) << endl;
        cout << getSize(c) << endl;
        cout << getSize(s) << endl;
        cout << getSize(i) << endl;
        cout << getSize(l) << endl;
        cout << getSize(f) << endl;
        cout << getSize(d) << endl;
        cout << getSize(g) << endl;
}
```

```
$ g++ getsize.cpp
$ a.out
1
1
2
4
8
4
8
16
```



Remarks

- The above illustrates generic programming.
- Instead of coding the function getSize() 8 times, one for each of the types bool, char, short, int, long long, float, double, long double, the programmer codes one function template with a parametrized type.
- The compiler instantiates the needed actual function by replacing the parametrized type with the actual type.

Two Function Templates Example (1)

```
$ cat functemp.cpp
#include <iostream>
#include <vector>                    // for vector<>
using namespace std;                 // for cout, endl, vector, ...
```

Two Function Templates Example (2)

```
class ratio {                                   // rational number
      int num, den;                             // numerator, denominator
public:
      ratio(int n=0, int d=1) {                 // 2-param constructor
            if(d<0)
                  { num = -n; den = -d; }       // den >= 0 always
            else
                  { num = n; den = d; }
      }
      bool operator < (const ratio & rhs) {
            return num*rhs.den < den*rhs.num;
      }
      friend ostream & operator << (ostream & lhs, const ratio & rhs);
};  // end of class ratio
```

Two Function Templates Example (3)

```
template <typename T>
void sortAny(vector<T> & v) {
      int i, j;
      for(i=0; i<v.size()-1; i++)
            for(j=i+1; j<v.size(); j++) {
                  if(v[j] < v[i]) {
                        T t = v[j];
                        v[j] = v[i];
                        v[i] = t;
                  }
            }
}
```

What is the sorting algorithm?

Two Function Templates Example (4)

```
template <typename T>
void showAny(const vector<T> & v) {
      cout << v[0];
      for(int i=1; i<v.size(); i++) cout << ' ' << v[i];
      cout << endl;
}
ostream & operator << (ostream & lhs, const ratio & rhs) {
    return lhs << rhs.num << '/' << rhs.den;
}
```

Two Function Templates Example (5)

```
int main() {
    vector<char> c;
    vector<int> i;
    vector<ratio> r;
    c.push_back('c'); c.push_back('s');
    c.push_back('1'); c.push_back('2');
    c.push_back('8'); c.push_back('0');
    i.push_back(1); i.push_back(2);
    i.push_back(8); i.push_back(0);
    r.push_back(ratio(1,0));
    r.push_back(ratio(1,-2));
    r.push_back(ratio(-8,0));
```

Two Function Templates Example (6)

```
    r.push_back(ratio());
    r.push_back(ratio(1)); r.push_back(ratio(2));
    r.push_back(ratio(8)); r.push_back(ratio(0));
    sortAny(c); showAny(c);
    sortAny(i); showAny(i);
    sortAny(r); showAny(r);
}
```

```
$ g++ functemp.cpp
$ a.out
0 1 2 8 c s
0 1 2 8
-8/0 -1/2 0/1 0/1 1/1 2/1 8/1 1/0
```

insertionSort Function Template

```
template <class T> void insertionSort (T a[], int n) {
     T val;
     int j;
    for (int i = 1; i < n; ++i) {
          if (a[i] < a[i-1]) {
             val = a[i];
             j = i;
             do {  a[j] = a[j-1];
                --j;
             } while ((j > 0) && (val < a[j-1]));
             a[j] = val;
          }
    }
}
```

Print array Function Template

```
template <typename T>
void print(ostream& out, T data[], int count) {
    out << "[ ";
    for (int i = 0; i < count; ++i) {
       out << data[i] << " ";
    }
    out << "]";
}
```

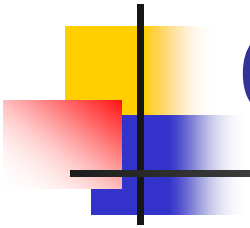
A pair of int

```
class pair {
private:
  int first;              // first value
  int second;             // second value

public:
  pair( int a, int b) : first(a), second(b) {}
  int get_first() const { return first; }
  int get_second() const { return second; }

};
```

This class can be used to handle a pair of int only



Class Templates

- Similar to function templates, C++ provides class templates.
- The compiler instantiates actual class declarations after replacing the parametried types and other parameters.

template<typename T> class pair

```
template<typename T>      // or template<class T>
class pair {
private:
  T first;                              // first value
  T second;                 // second value
public:
  pair(T a, T b) : first(a), second(b) {}
  T get_first() const                   { return first; }
  T get_second() const      { return second; }
};
```

Use of template<typename T> class pair

usage:

```
pair<int> int_pair(1, 2);     // must specify <int>
pair<string> str_pair("Hello", "World");
                                        // must specify <string>
```

Can we have
pair <int, string> int_string_pair(1,”satu”)?
Yes, we can if we define the template class pair with more than 1 type parameters

template <typename T1, typename T2>
class pair

```
template <typename T1, typename T2>
class pair {
private:
T1 first;
T2 second;
public:
      pair(T1 a, T2 b) : first(a), second(b) {}
      T1 get_first() const       { return first; }
  T2 get_second() const      { return second; }
};
```

STL provide this template class pair.

To use it, #include <utility>

Class Template Example (1)

```
$ cat classtmp.cpp
#include <iostream>
#include <vector>          // for vector<>
using namespace std;       // for cout, endl, vector, ...
template <typename T, int N>
class myArray {
    T v[N];
```

Class Template Example (2)

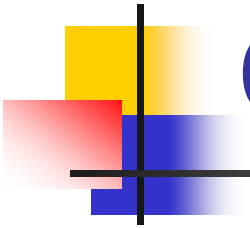
```
public:
    int getSize() { return N; }

    void setValue(const T & u, int i) { v[i] = u; }

    void show() {
        cout << v[0];
        for(int i=1; i<N; i++) cout << ' ' << v[i];
        cout << endl;
    }
```

Class Template Example (3)

```
void sort() {
      for(int i=0; i<N-1; i++)
            for(int j=i+1; j<N; j++)
                  if(v[j]<v[i]) {
                        T t = v[i];
                        v[i] = v[j];
                        v[j] = t;
                  }
      }
}; // end of template myArray
```



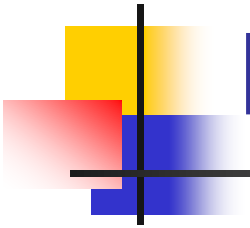
Class Template Example (4)

```
int main() {
    myArray<int,4> m1;
    myArray<float,3> m2;
    myArray<string,2> m3;
    m1.setValue(1,0);
    m1.setValue(2,1);
    m1.setValue(8,2);
    m1.setValue(0,3);
    m1.show();
    m1.sort();
    m1.show();
```

Class Template Example (5)

```
    m2.setValue(0.1,0);
    m2.setValue(0.02,1);
    m2.setValue(0.003,2);
    m2.show();
    m2.sort();
    m2.show();
    m3.setValue("cs",0);
    m3.setValue("1280",1);
    m3.show();
    m3.sort();
    m3.show();
}
```

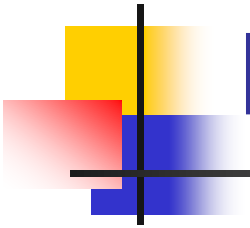
```
$ g++ classtmp.cpp
$ a.out
1 2 8 0
0 1 2 8
0.1 0.02 0.003
0.003 0.02 0.1
cs 1280
1280 cs
```



Chapter 8: Abnormal Control Flow

Outline:

- types of thrown items
- catching any item
- uncaught throws
- rethrowing
- standard exceptions



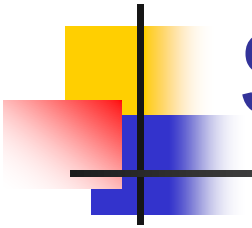
Exceptions

- Exceptions are errors that cause abnormal control flow.
- C++ provides the try-throw-catch mechanism for exception handling.

An Outline of try, throw, catch

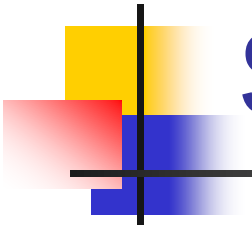
```
try {
        // some processing
        if( exception ) throw object;
        // here if exception did not happen
}
catch( Type1 p ) {
        // here if object is of Type1
}
catch( Type2 p ) {
        // here if object is of Type2
}
catch( ... ) {
        // here if object is none of the above types
}
```

The Try, Throw, Catch Semantics (1)



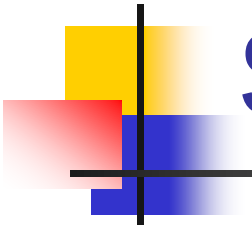
- In a try block, there can be one or more throw statements.
- A literal, variable, or object can be thrown.
- One or more catch blocks follow the try block.

The Try, Throw, Catch Semantics (2)



- The first catch block whose parameter type matches the type of the item thrown is executed, The thrown item initializes the parameter.
- The special catch(...) block with ellipsis as the parameter list catches any item thrown, regardless of its type.

The Try, Throw, Catch Semantics (3)



- If something has been thrown and caught, the catching block is executed and execution continues after skipping all other catch blocks.
- If nothing has been thrown, execution reaches the end of the try block and continues after skipping all the catch blocks.
- If something is thrown and not caught, the program is aborted.

Simple Try, Throw, Catch Example (1)

```
$ cat exc1.cpp
#include <iostream>
#include <cstdlib>
using namespace std;
class C{}; // to illustrate throwing object of this class
int main( int ac, char *av[] ) {
      int n;
      if(ac<2) n = -1;       // ac = 1 if a.out is entered
      else n = atoi(av[1]);
```

Simple Try, Throw, Catch Example (2)

```
try {
        switch(n) {
                case 0: throw (unsigned) 0;
                case 1: throw true;
                case 2: throw '2';
                case 3: throw (float) 3.0;   // it is double without casting
                case 4: throw 4.0;
                case 5: throw 5;
                case 6: throw "6";           // c-string, i.e., a char array
                case 7: throw (string) "7"; // cast a c-string to string
                case 8: throw C();           // create an unnamed object of class C
                default: cout << "throw nothing\t";
        }
} // end of try block
```

Simple Try, Throw, Catch Example (3)

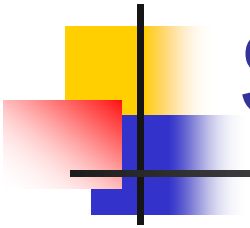
```
    catch(bool p) { cout << "bool\t\t" << p; }
    catch(char p) { cout << "char\t\t" << p; }
    catch(const char *p) { cout << "const char *\t" << p; }
    catch(double p) { cout << "double\t\t" << p; }
    catch(float p) { cout << "float\t\t" << p; }
    catch(int p) { cout << "int\t\t" << p; }
    catch(unsigned p) { cout << "unsigned\t" << p; }
    catch(...) { cout << "unknown\t\t" << '?'; }
    cout << "\treturning ..." << endl;
    return 0;
}
```

Simple Try, Throw, Catch Example (4)

```
$ g++ exc1.cpp
$ for i in -1 0 1 2 3 4 5 6 7 8 9
> do
> a.out $i
> done
```

```
throw nothing returning ...
unsigned 0 returning ...
bool 1 returning ...
char 2 returning ...
float 3 returning ...
double 4 returning ...
int 5 returning ...
const char * 6 returning ...
unknown ? returning ...
unknown ? returning ...
throw nothing returning ...
```

A Nested Function Throw and Stack Unwinding



- A throw from a nested function call originated from a try block is caught the same way.
- When this happens, the calling stack will be unwound, from the throwing function to the function containing the try block.
- That is, the stack frames of the function calls would be popped as if the functions are returning.
- Local objects of a function call would be destroyed as the stack frame of that function is popped.

Exceptions and Stack Unwinding Example (1)

```
#include <iostream>
using namespace std;
class peek {
public:
    peek(int p = 0): v(p) { cout << v << "\thi" << endl; }

    peek(const peek &p):v(-p.v) {cout<< v << "\tHi" <<endl;}

    ~peek() { cout << v << "\tbye" << endl; }
private:
    int v;
};
```

Exceptions and Stack Unwinding Example (2)

```
void f3(int p) {
        peek x(3);
        switch( p ) {
                case 0: throw (unsigned) 0;
                case 1: throw true;
                case 2: throw '2';
                case 3: throw (float) 3.0;
                case 4: throw 4.0;
                case 5: throw 5;
                case 6: throw "6";
                case 7: throw (string) "7";
                case 8: throw peek(x);          // call peek copy constructor
        }
        cout << "f3() returns" << endl;
}
```

Exceptions and Stack Unwinding Example (3)

```
void f2(int p) {
      peek x(2);
      f3(p);
      cout << "f2() returns" << endl;
}
void f1(int p) {
      peek x(1);
      f2(p);
      cout << "f1() returns" << endl;
}
```

Exceptions and Stack Unwinding Example (4)

```
int main( int ac, char *av[] ) {
    int n;
    if(ac < 2) n = -1; else n = atoi(av[1]);
    try {
        f1( n );
        cout << "try{} ends" << endl;
    }
```

Exceptions and Stack Unwinding Example (5)

```
    catch(bool p) { cout << "bool\t\t" << p << endl; }
    catch(char p) { cout << "char\t\t" << p << endl; }
    catch(const char *p){cout<<"const char *\t"<<p<< endl;}
    catch(double p) { cout << "double\t\t" << p << endl; }
    catch(float p) { cout << "float\t\t" << p << endl; }
    catch(int p) { cout << "int\t\t" << p << endl; }
    catch(unsigned p) { cout << "unsigned\t" << p << endl; }
    catch(...) { cout << "(...)\t" << '?' << endl; }
    cout << "main() returns" << endl;
    return 0;
}
```

Exceptions and Stack Unwinding Example (6)

```
main(){... try{... f1(n); ... } catch(){...} ... catch(){...}}
                        |
                        |
                        v
                        f2(p){...}
                        |
                        |
                        v
                        f3(p){
                        switch(p){
                        case 0: throw ...
                        ...
                        case 8: throw ...
                        }
...
```

Exceptions and Stack Unwinding Example (7)

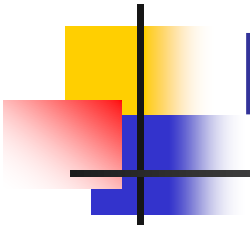
```
$ g++ fncall.cpp
$ a.out
1 hi
2 hi
3 hi
f3() returns
3 bye
f2() returns
2 bye
f1() returns
1 bye
try{} ends
main() returns
```

```
$ a.out 3
1 hi
2 hi
3 hi
3 bye
2 bye
1 bye
float 3
main() returns
```

```
$a.out 8
1       hi
2       hi
3       hi
-3      Hi
3       bye
2       bye
1       bye
(...)   ?
-3      bye
main() returns
```

float 3 is thrown in f3() but caught in main()

What will be printed for “a.out 7”?



Rethrowing an Exception

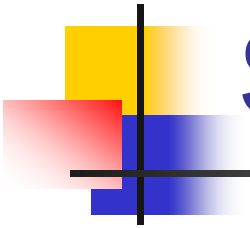
- An exception can be rethrown by the catch block that catches it.
- A rethrown exception is to be caught by the catch blocks of the next higher enclosing try block.

Rethrowing Example

```
#include <iostream>
using namespace std;
class C{};
int main() {
        try {
                try { throw C(); }
                catch(C p) { cout << "1st catch\n"; throw; }
                catch(...) { cout << "not here\n"; }
        }
        catch(C p) {cout << "2nd catch" << endl;}
        return 0;
}
```

```
$ g++ rethrow.cpp
$ a.out
1st catch
2nd catch
```

What if the rethrow is removed? Will 2nd catch be printed?



Standard Exceptions

- C++ standard library provides the base class std::exception so that components of the library can throw objects of various derived classes of exception.

<exception> & <stdexcept>

exception // <exception>

- logic_error // <stdexcept>
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error // <stdexcept>
  - overflow_error
  - range_error
  - underflow_error
- bad_alloc // <new>, thrown by new
- bad_cast // <typeinfo>, thrown by dynamic_cast
  // when fails with a referenced type
- bad_exception // <exception> thrown when an
  // exception doesn't match any catch
- bad_typeid // <typeinfo>, thrown by typeid
- ios_base::failure // <ios>, thrown by ios::clear

Exception example: domain_error

```
#include <algorithm>    // sort() declared
#include <stdexcept>
#include <vector>
template <class T> T median(vector<T> v){
    int size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    sort(v.begin(), v.end());
    int mid = size/2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```

User defined Exceptions (1)

```
// File: DivideByZeroException.h
using namespace std;
class DivideByZeroException {
  public:
        DivideByZeroException ()
        : message ("Divide By Zero Exception") { }

        const char * what () const { return message; }
  private:
        const char * message;
};
```

User defined Exceptions (2)

```
#include <iostream>
using namespace std;
int divide (int,int);
main () {
  int a, b;
  cout << "Enter a and b: ";
  cin >> a >> b;
  try {
    cout << "a/b = " << divide(a,b);
  } // end try
  catch (DivideByZeroException  e) {
     cout << e.what() << endl;
  } // end catch
}
```

```
int divide( int a, int b) {
if (b == 0)
throw DivideByZeroException ;
return a / b;

}
```

Enter a and b: 3 0

Divide By Zero Exception