Object Oriented Programming

In C++ (part 2)

Hakim Belhaouari
hakim.belhaouari@univ-poitiers.fr
(Thanks to Lilian Aveneau)

[1]University of Poitiers
Computer Science Department

[2]University Science and Technology of Ha Noi
USTH

2017-2018

Summary

1. Template in C++
   - Function template
   - Class template
   - Class template and inheritance

2. C++ Exception
   - Introduction: why?
   - Simple use
   - Advanced use
   - Programming with exceptions

Table of Contents

Table of Contents

Example

Function template creation

Function `min` generic : integer, real, ...

Example

Function template creation

Function `min` generic : integer, real, ...

```
template < typename T > // ou template<class T> old ''compilers
T min ( T a, T b ) {  return a<b ? a : b;  }
```

Example

Function template creation

Function `min` generic : integer, real, ...

```
template < typename T > // ou template<class T> old ''compilers
T min ( T a, T b ) {  return a<b ? a : b;  }
```

- Keywords `typename` (or `class`) : type parameter
- Acts like an "override", but managed by the compiler !
- Requires the same code for all instantiations.

Example

Function template creation

Function `min` generic : integer, real, ...

```
template < typename T > // ou template<class T> old ''compilers
T min ( T a, T b ) {  return a<b ? a : b;  }
```

- Keywords `typename` (or `class`) : type parameter
- Acts like an "override", but managed by the compiler !
- Requires the same code for all instantiations.

Use of function template

Compiler "builds" new functions (if needed)

```
int main () {
    const int n=4, p=12;
    cout << "min( n,p ) = " << min( n, p ) << endl; // need new function on int
    const float x=2.5f, y=3.25f;
    cout << "min( x,y ) = " << min( x, y ) << endl;
    const double lx=2.125, ly=3.6125;
    cout << "min(lx,ly) = " << min( lx, ly ) << endl;
    return 0;
}
```

The rule: the compiler needs the function body at compile time !

Be careful

The compiler checks all operators at instantiation time (ex: op "<" exists?)

Application au type char*

```
int main () {
    char*adr1 = (char*)"mister", *adr2 = (char*)"hello";
    cout <<"min(adr1, adr2) = " << min( adr1, adr2 ) << endl;
    return 0; // shows ''mister'' !
}
```

Be careful

The compiler checks all operators at instantiation time (ex: op "<" exists?)

Application au type char*

```
int main () {
    char*adr1 = (char*)"mister", *adr2 = (char*)"hello";
    cout <<"min(adr1, adr2) = " << min( adr1, adr2 ) << endl;
    return 0; // shows ''mister'' !
}
```

Instantiation with a class

```
class vecteur {
    int m_x, m_y;
public:
    vecteur( int x=0, int y=0 ) : m_x(x), m_y(y) {}
    void show() const { cout<<m_x<<" "<<m_y; }
    friend int operator<(const vecteur&a, const vecteur& b);
};
int operator<(const vecteur&a, const vecteur&b) {
    return a.m_x < b.m_x ? -1 : a.m_x > b.m_x ? +1
    : a.m_y < b.m_y ? -1 : a.m_y > b.m_y ? +1 : 0;
}
int main () {
    const vecteur u(3,2), v(4,1);
    const vecteur w = min(u, v);
    cout <<"min(u, v) = "; w.show(); cout << endl;
    return 0;
}
```

Type parameter and template definition

Each template may have one or many types parameters, which may be used anywhere in the function :

Type parameter and template definition

Each template may have one or many types parameters, which may be used anywhere in the function :

- Requires declaration inside a header
  - Allows instantiation of the function by the compiler.

Type parameter and template definition

Each template may have one or many types parameters, which may be used anywhere in the function :

- Requires declaration inside a header
  - Allows instantiation of the function by the compiler.
- For declaring local variables.

Type parameter and template definition

Each template may have one or many types parameters, which may be used anywhere in the function :

- Requires declaration inside a header
  - Allows instantiation of the function by the compiler.
- For declaring local variables.
- During executable instructions (*eg.* `new`, `sizeof`)

Type parameter and template definition

Each template may have one or many types parameters, which may be used anywhere in the function :

- Requires declaration inside a header
  - Allows instantiation of the function by the compiler.
- For declaring local variables.
- During executable instructions (*eg.* `new`, `sizeof`)

Example

```
template < typename T, typename U > T fct ( T a, T*b, U c ) {
   T x ;                // local variable, in the stack
   U * adr;             // again
   ...
   adr = new U[10]; // executable ``instruction'' , per compiler
   ...
   size_t n = sizeof( T ); // again
   ...
}
```

Identification type parameter for a function template

Strict instantiation rule (example: `min((int)i, (char)c);` will fail)

- No possible conversion : strict matching

```
char c; unsigned int q; int n, t[ 10 ], *adi;
const int ci1 = 10, ci2 =12; ...
min( n, c );      /// error : (T = int) != (T = char)
min( n, q );      /// error : (T = int) != (T = unsigned int)
min( n, ci1 );    /// error : const int and int mismatch
min( ci1, ci2 ); /// OK, where T == const int
min( t, adi );    /// OK ! where T = int*
```

Identification type parameter for a function template

Strict instantiation rule (example: `min((int)i, (char)c);` will fail)

- No possible conversion : strict matching

```
char c; unsigned int q; int n, t[ 10 ], *adi;
const int ci1 = 10, ci2 =12; ...
min( n, c );      /// error : (T = int) != (T = char)
min( n, q );      /// error : (T = int) != (T = unsigned int)
min( n, ci1 );    /// error : const int and int mismatch
min( ci1, ci2 );  /// OK, where T == const int
min( t, adi );    /// OK ! where T = int*
```

- User might give explicitly type parameter (exceptional conversion may occur)

```
template< typename T, typename U > T fct ( T x, U y, T z ) { return x+y+z; }
int main() {
int n = 1, p = 2, q=3;   float x = 2.5f, y = 5.f;
cout << fct( n, x, p ) << endl;   /// displays value (int) 5
cout << fct( x, n, y ) << endl;   /// displays value (float) 8.5
cout << fct( n, p, q ) << endl;   /// displays value (int) 6
cout << fct( n, p, x ) << endl;   /// ERROR : mismatch !
cout << fct<int,float> ( n, p, x ) << endl; /// displays (int)5
cout << fct<float> ( n, p, x ) << endl; /// displays (float)5.5, U is converted i
return 0;
}
```

Identification type parameter for a function template

Strict instantiation rule (example: `min((int)i, (char)c);` will fail)

- No possible conversion : strict matching

```
char c; unsigned int q; int n, t[ 10 ], *adi;
const int ci1 = 10, ci2 =12; ...
min( n, c );      /// error : (T = int) != (T = char)
min( n, q );      /// error : (T = int) != (T = unsigned int)
min( n, ci1 );    /// error : const int and int mismatch
min( ci1 , ci2 ); /// OK, where T == const int
min( t, adi );    /// OK ! where T = int*
```

- User might give explicitly type parameter (exceptional conversion may occur)

```
template< typename T, typename U > T fct ( T x, U y, T z ) { return x+y+z; }
int main() {
int n = 1, p = 2, q=3;    float x = 2.5f, y = 5.f;
cout << fct( n, x, p ) << endl;   /// displays value (int) 5
cout << fct( x, n, y ) << endl;   /// displays value (float) 8.5
cout << fct( n, p, q ) << endl;   /// displays value (int) 6
cout << fct( n, p, x ) << endl;   /// ERROR : mismatch !
cout << fct<int,float> ( n, p, x ) << endl; /// displays (int)5
cout << fct<float> ( n, p, x ) << endl; /// displays (float)5.5, U is converted i
return 0;
}
```

- Transmission mode : any

Syntax and limitations

Need pseudo constructors for standard type

Simplify some declarations :

```
template <typename T> void fct ( T a ) { T x(3); ... }
```

C++ adds pseudo-constructor for standard types :

```
double x( 3.5 ); // corresponds to    double x = 3.5;
char   c( 'e' ); // corresponds to    char   c = 'e';
```

Syntax and limitations

Need pseudo constructors for standard type

Simplify some declarations :

```
template <typename T> void fct ( T a ) { T x(3); ... }
```

C++ adds pseudo-constructor for standard types :

```
double x( 3.5 ); // corresponds to    double x = 3.5;
char   c( 'e' ); // corresponds to    char   c = 'e';
```

Limitations and properties

- Compatible with any types: user and basics
- Possible limitations:
  - Force at least one indirection `template<class T>void fct(T*) {... }`
  - Here implicitly allowed and so on ...

Syntax and limitations

Need pseudo constructors for standard type

Simplify some declarations :

```
template <typename T> void fct ( T a ) { T x(3); ... }
```

C++ adds pseudo-constructor for standard types :

```
double x( 3.5 ); // corresponds to    double x = 3.5;
char   c( 'e' ); // corresponds to    char   c = 'e';
```

Limitations and properties

- Compatible with any types: user and basics
- Possible limitations:
  - Force at least one indirection `template<class T>void fct(T*) {... }`
  - Here implicitly allowed and so on ...
- Limitations due to definition
  - `min` is not compatible with classes without definition of used operator: $<$
  - Same thing if constructor does not exist
  - ...

Expression parameter

Normal parameters may be used in parallel of classical `typename`

```
#include <iostream>
using namespace std;
// count the number of zero or equivalence...
template <typename T, int N> int count( T*tab) {
   int nz = 0;
   for (int i=0; i<N; i++) nz += !tab[i];
   return nz;
}
int main () {
   int  t[5] = { 5,2,0,2,0 };
   char c[6] = { 0,12,0,0,0,5 };
   cout << "count(t) = " << count<int,5>( t ) << endl;
   cout << "count(c) = " << count<char,6>( c ) << endl;
   return 0;
}
```

Expression parameter

Normal parameters may be used in parallel of classical `typename`

```
#include <iostream>
using namespace std;
// count the number of zero or equivalence...
template <typename T, int N> int count( T*tab) {
    int nz = 0;
    for (int i=0; i<N; i++) nz += !tab[i];
    return nz;
}
int main () {
    int  t[5] = { 5,2,0,2,0 };
    char c[6] = { 0,12,0,0,0,5 };
    cout << "count(t) = " << count<int,5>( t ) << endl;
    cout << "count(c) = " << count<char,6>( c ) << endl;
    return 0;
}
```

- Definition of function family ...

Expression parameter

Normal parameters may be used in parallel of classical `typename`

```
#include <iostream>
using namespace std;
// count the number of zero or equivalence...
template <typename T, int N> int count( T*tab) {
    int nz = 0;
    for (int i=0; i<N; i++) nz += !tab[i];
    return nz;
}
int main () {
    int  t[5] = { 5,2,0,2,0 };
    char c[6] = { 0,12,0,0,0,5 };
    cout << "count(t) = " << count<int,5>( t ) << endl;
    cout << "count(c) = " << count<char,6>( c ) << endl;
    return 0;
}
```

- Definition of function family ...
- Follow the same rules as normals functions

```
cout << "count(t) = " << count<int,5ul>( t ) << endl; // demo
cout << "count(c) = " << count<char,6.f>( c ) << endl; // demo
```

$\Longrightarrow$ Be careful of what you want and what you do...

template overridden

Examples

```
#include <iostream>
using std::cout; using std::endl;
template< typename T > T min (T a, T b)       { return a<b ? a : b; }
template< typename T > T min (T a, T b, T c) { return min( min( a, b ), c ); }
template< typename T > T min (T*tab, int n ) {
T m = tab[0]; for (int i=1; i<n; i++) m=min(m,tab[i]); return m; }
int main() {
   int n=12, p=15, q=2, t[5] = { 0, 6, 9, 3, -1 };
   float x=3.5f, y=4.25f, z=.25f;
   cout << min( n, p     ) << endl;
   cout << min( n, p, q ) << endl;
   cout << min( x, y, z ) << endl;
   cout << min( t, 5     ) << endl;
   return 0;
}
```

template overridden

Examples

```
#include <iostream>
using std::cout; using std::endl;
template< typename T > T min (T a, T b)      { return a<b ? a : b; }
template< typename T > T min (T a, T b, T c) { return min( min( a, b ), c ); }
template< typename T > T min (T*tab, int n ) {
T m = tab[0]; for (int i=1; i<n; i++) m=min(m,tab[i]); return m; }
int main() {
   int n=12, p=15, q=2, t[5] = { 0, 6, 9, 3, -1 };
   float x=3.5f, y=4.25f, z=.25f;
   cout << min( n, p    ) << endl;
   cout << min( n, p, q ) << endl;
   cout << min( x, y, z ) << endl;
   cout << min( t, 5    ) << endl;
   return 0;
}
```

Caution: cross satisfaction of function family

```
template< typename T > T min (T a, T b) { return  a< b ?  a :  b; }
template< typename T > T min (T*a, T b) { return *a< b ? *a :  b; }
template< typename T > T min (T a, T*b) { return  a<*b ?  a : *b; }
int main() {
   int n=12, p=15;
   float x=3.5f, y=4.25f;
   cout << min(  n,  p ) << endl; // 12  : version I   int min ( int, int )
   cout << min( &n,  p ) << endl; // 12  : version II  int min ( int*, int )
   cout << min(  x, &y ) << endl; // 3.5 : version III float min ( float, float* )
   cout << min( &x, &y ) << endl; // 0x7fff5fbff620 : I float* min ( float*, float* )
   return 0;
}
```

Function template specialization

Retake previous example on basic string

```
template< typename T > T min (const T& a, cont T& b) { return  a< b ?  a :  b; }
int main() {
   char *str1 = "mister";
   char *str2 = "hello";
   cout << min(  str1,  str2 ) << endl; /// "mister"
   return 0;
}
```

Function template specialization

Retake previous example on basic string

```
template< typename T > T min (const T& a, cont T& b) { return  a< b ?  a :  b; }
int main() {
   char *str1 = "mister";
   char *str2 = "hello";
   cout << min(  str1,  str2 ) << endl; /// "mister"
   return 0;
}
```

$\Longrightarrow$ We would like to specialize for this type only

Function template specialization

For this purpose the definition of a “classical” function is enough.

Example

```
#include <iostream>
#include <cstring> // strcmp
using namespace std;
// general definition
template<typename T> T min (const T& a, const T& b) {
   return a<b ? : a : b;
}
// specialized definition
char * min(char *a, char *b) {
   return strcmp(a,b) < 0 ? a : b;
}

int main() {
   int n = 12, p = 15;
   char *str1 = (char*)"mister", *str2 = (char*)"hello";
   cout << ::min(n,p)        << endl;
   cout << ::min(str1, str2) << endl;
   return 0;
}
```

NB : `::min` stands for our definition whereas `min` might correspond to `std::min`.

Partial specializations

A function family may include specialized ones which the compiler will choose wisely:

```
template <class T, class U> void fct (T a, U b) { ... } /// more general
template <class T>          void fct (T a, T b) { ... } /// more specific
```

Very useful for :

- Specific treatment when a pointer may occur:

```
template<class T> void f (T  t) { ... } // I
template<class T> void f (T* t) { ... } // II
...
int n, *adc;
f(  n  );  // version I
f( adi );  // version II, because more specific
```

- Distinguish between pointer and refenrece on a variable or a constant:

```
template < class T > void f( T& t) { ... } // I
template < class T > void f( const T& t) { ... } // II
...
int n; const int cn = 12;
f(  n ); // I where T == int
f( cn ); // II where T == int
```

Table of Contents

Class template example

```
#include <iostream>
using namespace std;
template<typename T> class point {
   T m_x, m_y;
public:
    point (T x, T y) : m_x(x), m_y(y) {}
    void display () const ;
};
template<typename T> /// informs compiler of template
void point<T>::display() const { /// the scope reminds parameters!
    cout << "Point(" << m_x << "," << m_y << ")" << endl;
}
int main () {
    point<float> A(1.5f, 2.f);
    point<int>   B(1, 2);
    A.display();
    B.display();
    return 0;
}
```

Class template example

```
#include <iostream>
using namespace std;
template<typename T> class point {
   T m_x, m_y;
public:
   point (T x, T y) : m_x(x), m_y(y) {}
   void display () const ;
};
template<typename T> /// informs compiler of template
void point<T>::display() const { /// the scope reminds parameters!
   cout << "Point(" << m_x << "," << m_y << ")" << endl;
}
int main () {
   point<float> A(1.5f, 2.f);
   point<int>   B(1, 2);
   A.display();
   B.display();
   return 0;
}
```

Constraint

- Class template must be “instantiable” per compiler
- As previously compiler must know completely the code
- Usually has a header file (file “.h”)

Type parameter

as much possible, unordered

```
template<typename T, typename U, typename V> // 3 types params
class essai {
   T x;                /// type x = T
   U t[5];             /// array of 5 element of type U
   V fml ( int , U );  /// funtion which returns type V and has 2 args (int*U)
};
```

Type parameter

as much possible, unordered

```
template<typename T, typename U, typename V> // 3 types params
class essai {
   T x;                    /// type x = T
   U t[5];                 /// array of 5 element of type U
   V fml ( int, U );       /// funtion which returns type V and has 2 args (int*U)
};
```

Instantiation a class template

```
essai< int, float, int > ce1;
essai< int, int*, double > ce2;
essai< char* int, obj > ce3;    // assume "obj" is a type!
essai< float, point<int>, point<float> > ce4;
```

Type parameter

as much possible, unordered

```
template<typename T, typename U, typename V> // 3 types params
class essai {
   T x;                 /// type x = T
   U t[5];              /// array of 5 element of type U
   V fml ( int , U );   /// funtion which returns type V and has 2 args (int*U)
};
```

Instantiation a class template

```
essai< int , float , int > ce1;
essai< int , int*, double > ce2;
essai< char* int , obj > ce3;    // assume "obj" is a type!
essai< float , point<int>, point<float> > ce4;
```

Remarks :

- No inference mechanism (explicit correlation)
- Instantiation has no side effect:

```
class essai { ...
template< typename T > void fct ( point<T> );
};
```

- Static member: each different instantiation owns one!

Static member: example

```cpp
// DEMO ?
#include <iostream>
using namespace std;

template < typename T >
class demo {
   T m_a;
   static int cpt;
public:
   demo( T a ) : m_a(a) {
      ++cpt;
      cout << "+++ " << cpt << " demo " << endl;
   }
   ~demo() {
      --cpt;
      cout << "--- " << cpt << " demo " << endl;
   }
   void affiche () const { cout << "demo( " << m_a << " )" << endl; }
};

/// a unique declaration that will be used for each instance
template< typename T > int demo<T>::cpt = 0;

int main () {
   demo<int>   ei1(1);    // +++ 1 demo
   demo<float> ef1(1.f);  // +++ 1 demo
   demo<float> ef2(2.f);  // +++ 2 demo
   demo<int>   ei2(2);    // +++ 2 demo
   return 0;               // --- 1 demo ...
}
```

Expression parameter

Usage:

- Any int numbers, use as all constant expressions

```
void foo(const int i) {    Demo<int, i> p;    p.affiche(); }
// fail unknow i at compile time
```

Expression parameter

Usage:

- Any int numbers, use as all constant expressions

```
void foo(const int i) {    Demo<int, i> p;    p.affiche(); }
// fail unknow i at compile time
```

- No conversion at instantiation time (`Demo<3> *p = new Demo<2>(); //fail`)
- No redefinition, then no more problem
- Usually, you can replace it by constructor parameter

Expression parameter

Usage:

- Any int numbers, use as all constant expressions

```
void foo(const int i) {    Demo<int, i> p;    p.affiche(); }
// fail unknow i at compile time
```

- No conversion at instantiation time (Demo<3> *p = new Demo<2>(); //fail)
- No redefinition, then no more problem
- Usually, you can replace it by constructor parameter

```
// DEMO?
#include <iostream>  /// watch out: academic example...
using namespace std;
template < typename T, int n > class array { T m_tab [ n ];
public:
   array() { cout << "array::constructor" << endl; }
   T& operator[] (int i) { return m_tab[ i ]; }
};
class point {  int m_x, m_y;
public:
   point( int x=0, int y=0 ) : m_x(x), m_y(y) { cout << "point::constructor"; affiche();  }
   void display() const { cout << "point( "<< m_x << ", "<< m_y << " )" << endl; }
};
int main () {
   tableau< int, 4 > ti;
   for ( int i=0; i<4; i++ ) ti[ i ] = i;
   for ( int i=0; i<4; i++ ) cout << ti[ i ] << " "; // display value from 0 to 3
   cout << endl;
   tableau< point, 3 > tp; // the point constructors are called (ordering)
   for ( int i=0; i<3; i++ ) tp[ i ].display();
   return 0; // destructors are called ...
}
```

Class template specialization

Example: specialization of one function member

```
#include <iostream>
using namespace std;
/// Encore notre bon vieux point
template< typename T > class point {
   T m_x, m_y;
public:
   point( T x=0, T y=0 ) : m_x(x), m_y(y) {}
   void display() const;
};
/// definition of the general code
template< typename T > void point<T>::display() const {
   cout << "point( " << m_x << ", " << m_y < " )" << endl;
}
/// definition of the specialized version dedicated to char
template<> void point<char>::display() const {
   cout << "point( " << (int)m_x << ", " << (int)m_y << " )" << endl;
}
/// tests
int main () {
   point< int    > ai(   3,   5 );   ai.display(); // point( 3, 5 )
   point< char   > ac( 'd', 'y' );   ac.display(); // point( 100, 121 )
   point< double > ad( 3.5, 2.3 );   ad.display(); // point( 3.5, 2.3 )
   return 0;
}
```

Here the specialization allows print the ASCII code of character

Different manners for specializing

1. One member function with all parameters

```
template< typename T, int n > class array {
   T m_tab[ n ];
public:
   array() { cout << "array::array" << endl; }
...
}
template<> array< point, 10 > :: array() {
   cout << "array::array<point,10>" << endl;
}
```

Different manners for specializing

1. One member function with all parameters

```
template< typename T, int n > class array {
   T m_tab[ n ];
public:
   array() { cout << "array::array" << endl; }
...
}
template<> array< point, 10 > :: array() {
   cout << "array::array<point,10>" << endl;
}
```

2. One specialized class: in this case, you can specialize all member functions

```
template<> class point<char> {
   char m_x, m_y;
public:
   point( char x=0, char y=0 ) : m_x(x), m_y(y) {}
   void display() const;
};
void point<char>::display() const {
   cout << "point( " << (int)m_x << ", " << (int)m_y << " )" << endl;
}
```

Different manners for specializing

1. One member function with all parameters

```
template< typename T, int n > class array {
   T m_tab[ n ];
public:
   array() { cout << "array::array" << endl; }
...
}
template<> array< point , 10 > :: array() {
   cout << "array::array<point,10>" << endl;
}
```

2. One specialized class: in this case, you can specialize all member functions

```
template<> class point<char> {
   char m_x, m_y;
public:
   point( char x=0, char y=0 ) : m_x(x), m_y(y) {}
   void display() const;
};
void point<char>::display() const {
   cout << "point( " << (int)m_x << ", " << (int)m_y << " )" << endl;
}
```

3. Partial specification is able for function member and class declaration with the syntax.

Partial specialization and default parameters

Example:

```
template< typename T, typename U > class A          { .... } /// version I
template< typename T >             class A< T, T*> { .... } /// version II
```

Here `A< int, float > a1;` uses version I

But `A< int, int*> a2;` uses version II

Partial specialization and default parameters

Example:

```
template< typename T, typename U > class A          { .... } /// version I
template< typename T >             class A< T, T*> { .... } /// version II
```

Here `A< int, float > a1;` uses version I

But `A< int, int*> a2;` uses version II

Default value for type parameters

Similar to function template

```
template< typename T, typename U=float > class A { ... } ;
template< typename T, int n=3 >          class B { ... } ;
...
A< int, long > a1; /// normal behavior
A< int >       a2; //  means "A< int, float > a2;"
B< int, 3 >    b1; /// normal behavior
B< int >       b1; // means "B< int, 3 > b2;"
```

Be careful: no meaning for member functions

member function template and identity

member functions template

- Similar to the member function in ordinary class

```
class A { ...
template< typename T > void fct ( T ) ;
...
};
```

- and for member function of class template

```
template< typename T > class A { ....
template< typename U > void fct ( U x );
...
};
```

member function template and identity

member functions template

- Similar to the member function in ordinary class

```
class A { ...
template< typename T > void fct ( T ) ;
...
};
```

- and for member function of class template

```
template< typename T > class A { ....
template< typename U > void fct ( U x );
...
};
```

Identity of class template

Example :

```
array< int   , 12 > t1;
array< float , 12 > t2;
...
t2 = t1; // incorrect (different values)
```

Same if sizes are different, ...

Same classes iff all parameters are the same

Class template and friendship

- Classes or "classical" friend function

```
template< typename T > class test {
    int m_x;
public:
    friend class A;            /// A see private attr. of ALL instantiation of test
    friend int fct( float ); // fct() is friend with ALL instantiation of test
    ...
};
```

Class template and friendship

- Classes or "classical" friend function

```
template< typename T > class test {
    int m_x;
public:
    friend class A;              /// A see private attr. of ALL instantiation of test
    friend int fct( float );  // fct() is friend with ALL instantiation of test
    ...
};
```

- Particular instances of class/function template
    - Be the class: point<T>
    - and the function: `template<class T> int fct( T x);` ⇒ we have 2 cases:

```
template<class T, class U> class essai1
{ int m_x;
public: /// friendship with all instances
    friend class point<int>;
    friend int fct(double);
    ...
};
```

```
template<class T, class U> class essai2
{  int m_x;
public: /// ``coupling''
    friend class point<T>;
    friend int fct( U ) ;
    ...
};
```

Class template and friendship

- Classes or "classical" friend function

```
template< typename T > class test {
    int m_x;
public:
    friend class A;          /// A see private attr. of ALL instantiation of test
    friend int fct( float ); // fct() is friend with ALL instantiation of test
    ...
};
```

- Particular instances of class/function template
    - Be the class: `point<T>`
    - and the function: `template<class T> int fct( T x);` ⇒ we have 2 cases:

```
template<class T, class U> class essai1
{ int m_x;
public: /// friendship with all instances
    friend class point<int>;
    friend int fct(double);
    ...
};
```

```
template<class T, class U> class essai2
{  int m_x;
public: /// ''coupling''
    friend class point<T>;
    friend int fct( U ) ;
    ...
};
```

- With another class/function template

```
template< typename T, typename U > class essai3 {
    int m_x;
public: /// all instances are friends with all instances
    template< typename X > friend class point< X >;
    template< typename X > friend int fct ( point< X > );
    ... /// same thing for functions
};
```

Example : 2D array (1/2)

Be the followed class:

```
template < typename T, int n > class array {
   T m_tab[ n ];
public:
   T& operator [] ( int i ) { return m_tab[ i ] ; }
}
```

Then we can declare:

```
array< array< int , 2 >, 3 > t2d;
```

The expression `t2d[1][2]` accesses to third element of the second array.

Let's take a more realist example:

- we handle the out of bounds
- we initialize the elements

```
#include <iostream>
using namespace std;
template< typename T, int n > class array {
   T    m_tab[ n ];
   int  m_limit;
public:
   array ( const T& init=0 ) { // assumes constructor on T
   for (int i=0; i<n; i++ ) m_tab[ i ] = init;
   m_limit = n-1;
   cout << "+++ array<T,"<<n<<">("<<init<<")"<<endl;
}
```

Example : 2D array (2/2)

```
T& operator[] ( int i ) {
   if ( i<0 || i>m_limit ) {
      cout << "-- out of bounds : " << i << endl;
      i = 0; /// trick: better solution with exception
   }
   return m_tab[ i ];
}
/// example
friend ostream& operator << ( ostream& out, array<T,n> t ) {
   out<<"[";
   for (int i=0; i<t.m_limit; i++ ) out << t.m_tab[ i ] << ",";
      if ( t.m_limit > -1 ) out << t.m_tab[ t.m_limit ];
      return out<<"]";
   }
};
/// tests ...
int main () {
   array< array< int, 3 >, 2 > ti;
   array< array< float, 4 >, 2 > td ( 10.f );
   ti[ 1 ][ 6 ] = 15;
   td[ 8 ][-1 ] = -1.f;
   cout << ti << endl;
   cout << td << endl;
   return 0;
}
/// Display :
/// +++ array<T,3>(0)
/// +++ array<T,3>(0)            ... +++ array<T,2>([10,10,10,10])
/// +++ array<T,3>(0)            ... -- out of bounds : 6
/// +++ array<T,2>([0,0,0])      ... -- out of bounds : 8
/// +++ array<T,4>(10)           ... -- out of bounds : -1
/// +++ array<T,4>(0)            ... [[0,0,0],[15,0,0]]
/// +++ array<T,4>(0)            ... [[-1,10,10,10],[10,10,10,10]]
```

Table of Contents

"Ordinary" class inherit a class template

Example : `class B : public A< int >`

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

template< typename T > class Point {
   T m_x, m_y ;
public:
   Point( T a=0, T o=0 ) : m_x(a), m_y(o) {}
   void display() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
};

class PointCol_int : public Point<int> {
   short m_color;
public:
   PointCol_int( int a=0, int o=0, short c=1 ) : Point<int>( a, o ), m_color(c) {}
   void display() {
      Point<int>::display();
      cout<<"\tcolor : "<<m_color<<endl;
   }
};

int main() {
   Point<float> pf ( 3.5, 2.8 ); pf.display();
   PointCol_int p ( 3, 5, 9 );   p.display();
   return 0;
}
```

$\Longrightarrow$ No problem ...

Inheritance from same parameters template

Example : `template< typename T > class B : public A< T >`

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

template< typename T > class Point {
    T m_x, m_y ;
public:
    Point( T a=0, T o=0 ) : m_x(a), m_y(o) {}
    void display() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
};

template< typename T > class PointCol : public Point<T> {
    short m_color;
public:
    PointCol( T a=0, T o=0, short color=1 ) : Point<T>( a, o ), m_color(color) {}
    void display() {
        Point<T>::display();
        cout<<"\tcolor : "<<m_color<<endl;
    }
};

int main() {
    Point<float> pf ( 3.5, 2.8 ); pf.display();
    PointCol<int> p ( 3, 5, 9 );  p.display();
    return 0;
}
```

Inheritance with new type parameters

Example :

`template<class T, class U> class B : public class A<T>`

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

template< typename T > class Point {
   T m_x, m_y ;
public:
   Point( T a=0, T o=0 ) : m_x(a), m_y(o) {}
   void display() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
};

template< typename T, typename U > class PointCol : public Point<T> {
   U m_color;
public:
   PointCol( T a=0, T o=0, U color=1 ) : Point<T>( a, o ), m_color(color) {}
   void display() {
   Point<T>::display();
   cout<<"\tcolor : "<<m_color<<endl;
   }
};

int main() {
   Point<float> pf ( 3.5, 2.8 );        pf.display();
   PointCol<int,short> p ( 3, 5, 9 );   p.display();
   return 0;
}
```

Table of Contents

Table of Contents

Errors managements

The old way (coming from C: checking all functions):

1. Global dedicated variable (Linux: `errno` and `perror()`)
    - Very tedious, in practice nobody do it
2. Use specific mechanism of the OS (ex: raison a signal through the function `raise()`
3. Hack the standard call of method with specific functions (`setjmp()` and `longjmp()`)

Errors managements

The old way (coming from C: checking all functions):

1. Global dedicated variable (Linux: `errno` and `perror()`)
   - Very tedious, in practice nobody do it
2. Use specific mechanism of the OS (ex: raison a signal through the function `raise()`
3. Hack the standard call of method with specific functions (`setjmp()` and `longjmp()`)

The two last methods prevent C++ mechanisms (ex: call of destructors)

Example

```
// DEMO
#include <iostream>
#include <csetjmp>
#include <cstdio> // rand
using namespace std;
class Array { int *tab; uint size;
public:
      Array(uint s) : size(s) { tab = new int[size]; cout<<"Constructor"<<endl; }
      ~Array() { cout<<"BEGIN_Destructor"<<endl; delete[] tab; cout<<"END_Destructor"<<endl;}
};
jmp_buf myBackup; // structure for memoring execution context
void f() {
   Array r(10);
   cout<<"Declaration_of_my_local_variable"<<endl;
   double luck = ((double)rand()/ (double)RAND_MAX);
   if(luck < 0.5)
      longjmp(myBackup, 1); // any value excepted 0
}
int main() {
   if(setjmp(myBackup) == 0) {
      cout<<"I_do_my_treatment"<<endl;
      f();
   }
   else {
      cout<<"Context_restored"<<endl;
   }
   cout<<"END:_do_I_clear_correctly_my_memory?"<<endl;
}
```

Example

```
// DEMO
#include <iostream>
#include <csetjmp>
#include <cstdio> // rand
using namespace std;
class Array { int *tab; uint size;
public:
      Array(uint s) : size(s) { tab = new int[size]; cout<<"Constructor"<<endl; }
      ~Array() { cout<<"BEGIN Destructor"<<endl; delete[] tab; cout<<"END Destructor"<<endl;}
};
jmp_buf myBackup; // structure for memoring execution context
void f() {
   Array r(10);
   cout<<"Declaration of my local variable"<<endl;
   double luck = ((double)rand()/ (double)RAND_MAX);
   if(luck < 0.5)
      longjmp(myBackup, 1); // any value excepted 0
}
int main() {
   if(setjmp(myBackup) == 0) {
      cout<<"I do my treatment"<<endl;
      f();
   }
   else {
      cout<<"Context restored"<<endl;
   }
   cout<<"END: do I clear correctly my memo
}
```

Display when luck >= 0.5

```
$ ./demoSetjmp
I do my treament
Constructor
Declaration of my local variable
BEGIN Destructor
END Destructor
END: do I clear correctly my memory?
```

Example

```
// DEMO
#include <iostream>
#include <csetjmp>
#include <cstdio> // rand
using namespace std;
class Array { int *tab; uint size;
public:
      Array(uint s) : size(s) { tab = new int[size]; cout<<"Constructor"<<endl; }
      ~Array() { cout<<"BEGIN Destructor"<<endl; delete[] tab; cout<<"END Destructor"<<endl;}
};
jmp_buf myBackup; // structure for memoring execution context
void f() {
   Array r(10);
   cout<<"Declaration of my local variable"<<endl;
   double luck = ((double)rand()/ (double)RAND_MAX);
   if(luck < 0.5)
      longjmp(myBackup, 1); // any value exce
}
int main() {
   if(setjmp(myBackup) == 0) {
      cout<<"I do my treatment"<<endl;
      f();
   }
   else {
      cout<<"Context restored"<<endl;
   }
   cout<<"END: do I clear correctly my memo
}
```

Display when luck < 0.5

```
$ ./demoSetjmp
I do my treament
Constructor
Declaration of my local variable
Context restored
END: do I clear correctly my memory?
```

```
$ ./demoSetjmp
I do my treament
Constructor
Declaration of my local variable
BEGIN Destructor
END Destructor
END: do I clear correctly my memory?
```

Handling exception in C++

Two main advantages:

1. Handling exception is easier:
    - Develop code as usual ...
    - On a separate bloc, you handle the error.
    - You may regroup dangerous instructions.

2. Exceptions must not be ignored:
    - Error : it is an object.
    - The developer must catch a thrown object
    - The first executed error block depends on the call stack.
    - In worst case, the compiler make a default catcher around the entry point.

Table of Contents

Throw an exception with keyword: throw

```
class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data(msg) {}
};

void f() {
  // Ici on "lance" un objet exception
  throw MyError("something bad happened");
}

int main() {
  /// Ici on aimerait l'attraper ... (coming soon)
  f();
}
```

Result

```
$ ./MyError
terminate called after throwing an instance of 'MyError'
Abort trap
```

Throw an exception with keyword: `throw`

```
class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data(msg) {}
};

void f() {
  // Ici on "lance" un objet exception
  throw MyError("something bad happened");
}

int main() {
  /// Ici on aimerait l'attraper ... (coming soon)
  f();
}
```

Result

```
$ ./MyError
terminate called after throwing an instance of 'MyError'
Abort trap
```

1. Create a new object of class `MyError`, and thrown it
2. Stack unwinding : destroy local objects to the block
3. Any objects/pointers can be used as error ...

How do I catch exception?

Two steps:

First: delimit dangerous area with `try` block

```
try {
// dangerous instructions
// that may throw many exceptions
}
```

How do I catch exception?

Two steps:

First: delimit dangerous area with `try` block

```
try {
// dangerous instructions
// that may throw many exceptions
}
```

Second: list all catchable exceptions and they code with `catch` block

```
try {
/// Code that may generate exceptions
} catch(type1 id1) {
/// Handle exceptions of type1
} catch(type3 id3)
/// Etc...
} catch(typeN idN)
/// Handle exceptions of typeN
}
// Normal execution resumes here...
```

How do I catch exception?

Two steps:

First: delimit dangerous area with `try` block

```
try {
// dangerous instructions
// that may throw many exceptions
}
```

Second: list all catchable exceptions and they code with `catch` block

```
try {
/// Code that may generate exceptions
} catch(type1 id1) {
/// Handle exceptions of type1
} catch(type3 id3)
/// Etc...
} catch(typeN idN)
/// Handle exceptions of typeN
}
// Normal execution resumes here...
```

- If the try block finished correctly you continue the normal execution
- Only one catch block can be executed for one try block
- The polymorphism is allowed for regrouping exceptions codes

Exemple 1

```
#include <iostream>
#include <cstdlib> /// pour exit(), EXIT_FAILURE ou SUCCESS
using namespace std;
/// Indiquer une erreur d'indice
class VecteurLimite {
  const int pos;
public:
  VecteurLimite( int i ) : pos(i) {}
  int whichPos() const { return pos; }
};
/// Vecteur classique, version patron
template<typename T> class Vecteur {
  const int nelem;
  T*const adr;
public:
  Vecteur(int n) : nelem(n), adr(new T[n]) {}
  ~Vecteur() { cout<<"Vecteur destructeur ..."<<endl; delete[] adr; }
  T& operator[] (int i) {
    if ( i<0 || i>=nelem ) { VecteurLimite vl(i); throw vl; } // levee d'exception
    return adr[i];
  }
};
/// Fonction principale ...
int main() {
  try {
    Vecteur<int> v(10); // destructeur sera bien appele ...
    for (int i=0; i<20; i++) { v[10] = 5; }
  }
  catch (VecteurLimite vl) {
    cout << "exception limite en " << vl.whichPos() <<endl;
    exit( EXIT_FAILURE );
  };
  return EXIT_SUCCESS;
}
```

Exemple 2

demo: what happen if we put a try in a try block with many local variables at each block?

Regrouping exceptions

The polymorphism is able only through pointer and reference...

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

/// Classe de base ...
class Except1 {};

/// Classe derivee ...
class Except2 {
public:
  Except2(const Except1&) {}
};

/// fonction sans interet, leve exception
void f() { throw Except1(); }

/// fonction principale
int main() {
  try {
    f();
  } catch(Except2&) {
    // et non, pas de conversion ...
    cout << "inside catch(Except2)" << endl;
  } catch(Except1&) {
    // ce bloc traitera l'exception ...
    cout << "inside catch(Except1)" << endl;
  }
}
```

Regrouping exceptions

The polymorphism is able only through pointer and reference...

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

/// Classe de base ...
class Except1 {};

/// Classe derivee ...
class Except2 {
public:
  Except2(const Except1&) {}
};

/// fonction sans interet, leve exception
void f() { throw Except1(); }

/// fonction principale
int main() {
  try {
    f();
  } catch(Except2&) {
    // et non, pas de conversion ...
    cout << "inside catch(Except2)" << endl;
  } catch(Except1&) {
    // ce bloc traitera l'exception ...
    cout << "inside catch(Except1)" << endl;
  }
}
```

```
#include <iostream>
using namespace std;
/// Definition hierarchie d'exceptions ...
class X {
public:
  class Trouble {}; /// La racine
  class Small : public Trouble {};
  class Big : public Trouble {};
   /// fonction utilitaire
  void f() { throw Big(); }
};

/// Fonction principale
int main() {
  X x;
  try {
    x.f();
  } catch(X::Trouble&) {
    // Ce gestionnaire sera appele ...
    cout << "caught Trouble" << endl;
    // Masquee par gestionnaire precedent
  } catch(X::Small&) { /// Warning ...
    cout << "caught Small Trouble" << endl;
  } catch(X::Big&) {
    cout << "caught Big Trouble" << endl;
  }
}
```

Bad ordering changes the scope of visible exception handlers

Cath all and Re-throw

Catch all

```
catch (...) { // usually in last position
cout << "an anonymous exception has been thrown" << endl;
}
```

Cath all and Re-throw

Catch all

```
catch (...) { // usually in last position
cout << "an anonymous exception has been thrown" << endl;
}
```

OK, but what can I do with that?

Cath all and Re-throw

Catch all

```
catch (...) { // usually in last position
cout << "an anonymous exception has been thrown" << endl;
}
```

OK, but what can I do with that?

Re-throw

```
catch (...) {
cout << "an anonymous exception has been thrown" << endl;
/// make treatment about before the try or anything else (close a file, ...)
throw; // re-throw the exception FROM THIS block (and may be un-masked)
}
```

Uncatched exceptions...

The default catcher is called automatically and it calls `terminate()` (which contains a call to `abort()`)

- Destructor may not be called (memory leaks)

```
#include <exception> /// contient "set_terminate"
#include <iostream>
using namespace std;

void terminator() { /// nouvelle version
  cout << "I'll be back!" << endl;
  exit(0); /// plus "propre" que abort()
}

void (*old_terminate)() = set_terminate(terminator);

class Botch { /// Bousiller
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; } // beurk !
};

int main() {
  try {
    Botch b;
    b.f();
  } catch(...) { // Et non, on ne va pas la, car double levee d'exception ...
    cout << "inside catch(...)" << endl;
```

To be clear:

When a constructor is done, you MUST call its destructor ...

```
#include <iostream>
using namespace std;
/// Classe basique, avec compteur partage
class Trace {
  static int counter; /// Variable de classe
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid << endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid << endl;
  }
};
/// Rappel : initialisation des variables de classe
int Trace::counter = 0;
/// Fonction principale : construire des instances
int main() {
  try {
    Trace n1;
    Trace array[5]; // va lever une exception
    Trace n2;  // donc on n'ira pas ici ...
  } catch(int i) {
    cout << "caught " << i << endl;
  }
}
```

To be clear:

When a constructor is done, you MUST call its destructor

```cpp
#include <iostream>
using namespace std;
/// Classe basique, avec compteur partage
class Trace {
  static int counter; /// Variable de classe
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid << endl;
  }
};
/// Rappel : initialisation des variables de classe
int Trace::counter = 0;
/// Fonction principale : construire des instances
int main() {
  try {
    Trace n1;
    Trace array[5]; // va lever une exception
    Trace n2;  // donc on n'ira pas ici ...
  } catch(int i) {
    cout << "caught " << i << endl;
  }
}
```

Display

```
$ ./Cleanup
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3
```

Table of Contents

Memory discussions

When an exception is thrown, do all resources correctly free?

- Hard task: when the constructor fails before its end.

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

class Cat {
public:
    Cat()  { cout << "Cat()" << endl; }
    ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
    void* operator new(size_t sz) {
        cout << "allocating a Dog["
        << sz << "]" << endl;
        throw 47;
    }
    void operator delete(void* p) {
        cout << "deallocating a Dog" << endl;
        ::operator delete(p);
    }
};
```

```cpp
class UseResources {
    Cat* bp;
    Dog* op;
public:
    UseResources(int count = 1) {
        cout << "UseResources()" << endl;
        bp = new Cat[count];
        op = new Dog; // 47
    }
    ~UseResources() {
        cout << "~UseResources()" << endl;
        // Array delete, don't be called
        delete [] bp;
        delete op;
    }
};
int main() {
    try {
        UseResources ur(3);
    } catch(int) {
        cout << "inside handler" << endl;
    }
}
```

Memory discussions

When an exception is thrown, do all resources correctly free?

- Hard task: when the constructor fails before its end.

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

class Cat {
public:
    Cat()  { cout << "Cat()" << endl; }
    ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
    void* operator new(size_t sz) {
        cout << "allocating a Dog["
       << sz << "]" << endl;
        throw 47;
    }
    void operator delete(void* p) {
        cout << "dealloca
        ::operator delete
    }
};
```

```
class UseResources {
    Cat* bp;
    Dog* op;
public:
    UseResources(int count = 1) {
        cout << "UseResources()" << endl;
        bp = new Cat[count];
        op = new Dog; // 47
    }
    ~UseResources() {
        cout << "~UseResources()" << endl;
        // Array delete, don't be called
        delete [] bp;
        delete op;
    }
};
int main() {
    try {
                              dler" << endl;
```

Display

```
$ ./Cleanup
UseResources()
Cat()
Cat()
Cat()
allocating a Dog[1]
inside handler
```

Memory discussions

When an exception is thrown, do all resources correctly free?

- Hard task: when the constructor fails before its end.

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

class Cat {
public:
    Cat()  { cout << "Cat()" << endl; }
    ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
    void* operator new(size_t sz) {
        cout << "allocating a Dog["
         << sz << "]" << endl;
        throw 47;
    }
    void operator delete(void* p) {
        cout << "dealloca
        ::operator delete
    }
};
```

```
class UseResources {
    Cat* bp;
    Dog* op;
public:
    UseResources(int count = 1) {
        cout << "UseResources()" << endl;
        bp = new Cat[count];
        op = new Dog; // 47
    }
    ~UseResources() {
        cout << "~UseResources()" << endl;
        // Array delete, don't be called
        delete [] bp;
        delete op;
    }
};
int main() {
    try {

                     dler" << endl;
```

Display

```
$ ./Cleanup
UseResources()
Cat()
Cat()
Cat()
allocating a Dog[1]
inside handler
```

Solution : use only object

1. Simple solution: catch exception in the constructor `UserResources`
2. RAII : Resource Acquisition Is Initialization design pattern from Bjarne Stroustrup

```cpp
#include <iostream>
#include <cstddef>
using namespace std;
///Simplified. Yours may have other arguments
template<class T, int sz = 1> class PWrap {
   T* ptr;
public:
class RangeError {}; /// Exception class
   PWrap() {
      ptr = new T[sz];
      cout << "PWrap constructor" << endl;
   }
   ~PWrap() {
      delete[] ptr;
      cout << "PWrap destructor" << endl;
   }
   T& operator[](int i) throw(RangeError) {
      if(i >= 0 && i < sz) return ptr[i];
      throw RangeError();
   }
};

class Cat { /// Almost the same
public:
   Cat() { cout << "Cat()" << endl; }
   ~Cat() { cout << "~Cat()" << endl; }
   void g() {}
```

```cpp
class Dog { /// idem
public:
   void* operator new[](size_t) {
      cout << "Allocating a Dog" << endl;
      throw 47;
   }
   void operator delete[](void* p) {
      cout << "Deallocating a Dog" << endl;
      ::operator delete[](p);
   }
};
class UseResources { /// New version
   PWrap<Cat, 3> cats; // all is here
   PWrap<Dog> dog;      // and here
public:
   UseResources() {
      cout << "UseResources()" << endl; }
   ~UseResources() {
      cout << "~UseResources()" << endl; }
   void f() { cats[1].g(); }
};
int main() {
   try {
      UseResources ur;
   } catch(int) {
      cout << "inside handler" << endl;
   } catch(...) {
```

Solution : use only object

1. Simple solution: catch exception in the constructor `UserResources`
2. RAII : Resource Acquisition Is Initialization design pattern from Bjarne Stroustrup

```cpp
#include <iostream>
#include <cstddef>
using namespace std;
///Simplified. Yours may have other argumen
template<class T, int sz = 1> class PWrap {
   T* ptr;
public:
class RangeError {}; /// Exception class
   PWrap() {
      ptr = new T[sz];
      cout << "PWrap constructor" << endl;
   }
   ~PWrap() {
      delete[] ptr;
      cout << "PWrap destructor" << endl;
   }
   T& operator[](int i) throw(RangeError) {
      if(i >= 0 && i < sz) return ptr[i];
      throw RangeError();
   }
};

class Cat { /// Almost the same
public:
   Cat() { cout << "Cat()" << endl; }
   ~Cat() { cout << "~Cat()" << endl; }
   void g() {}
```

Display

```
$ ./Wrapped
Cat()
Cat()
Cat()
PWrap constructor
Allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

```cpp
public:
   UseResources() {
      cout << "UseResources()" << endl; }
   ~UseResources() {
      cout << "~UseResources()" << endl; }
   void f() { cats[1].g(); }
};
int main() {
   try {
      UseResources ur;
   } catch(int) {
      cout << "inside handler" << endl;
   } catch(...) {
```

Standard : auto_ptr

RAII : includes directly in the stl.

- modifies operator $*$ et $\rightarrow$
- Example

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

class TraceHeap {
  int i;
public:
  static void* operator new(size_t siz) {
    void* p = ::operator new(siz);
    cout << "Allocating TraceHeap object on the heap "
         << "at address " << p << endl;
    return p;
  }
  static void operator delete(void* p) {
    cout << "Deleting TraceHeap object at address "
         << p << endl;
    ::operator delete(p);
  }
  TraceHeap(int i) : i(i) {}
  int getVal() const { return i; }
};

int main() {
  auto_ptr<TraceHeap> pMyObject( new TraceHeap(5) ); // affiche message constructeur
  cout << pMyObject->getVal() << endl;  // Affiche 5
} // destructeur est bien appele (verifiez message affiche) !
```

Standard exceptions

The header <stdexcept> includes the header <exception>. They defines common usely exceptions:

- `class exception` : the root, has one method `what()`
  1. `class bad_exception : public exception` : bad arguments, *etc.*
  2. `class logic_error : public exception` : error that should be detected at compile time.
  3. `class runtime_error : public exception` : error that could be detected at execution time.

Standard exceptions

The header <stdexcept> includes the header <exception>. They defines common usely exceptions:

- `class exception` : the root, has one method `what()`
  1. `class bad_exception : public exception` : bad arguments, *etc.*
  2. `class logic_error : public exception` : error that should be detected at compile time.
  3. `class runtime_error : public exception` : error that could be detected at execution time.

Example

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

class MyError : public runtime_error {
public:
  MyError(const string& msg = "") : runtime_error(msg) {}
};

int main() {
  try {
    throw MyError("my_message");
  } catch(MyError& x) {
    cout << x.what() << endl;
  }
```

Derived exceptions

Derived exception of `logic_error`

`domain_error`	precondition violation
`invalid_argument`	Invalid argument for a function
`length_error`	Indicates an attempt to produce a too big object (greater than a specific size)
`out_of_range`	Indicates argument out of range
`bad_cast`	When a bad `dynamic_cast` occurs
`bad_typeid`	Occurs when a null pointer is used on the `typeid(*p)`

Derived exception of `runtime_error`

`range_error`	Post-condition violation
`overflow_error`	Report an arithmetic overflow
`bad_alloc`	Throw by operator `new` when there is not enough memory.

Table of Contents

throwable exception in signature

Not required in C++ but **strongly** recommended ...

```
void f() throw (tooBig, tooSmall, divZero);
```

Attention: nothing means everything...

```
void f() ;          // Every exceptions may occur
void f() throw (); // No exception can be thrown
```

throwable exception in signature

Not required in C++ but **strongly** recommended ...

```
void f() throw (tooBig, tooSmall, divZero);
```

Attention: nothing means everything...

```
void f() ;          // Every exceptions may occur
void f() throw (); // No exception can be thrown
```

Useful for destructor!

```
class exception {
public:
    exception() throw() { }
    virtual ~exception() throw();
    /// Returns C-style character string describing the general cause of the current error
    virtual const char* what() const throw();
};
```

Inheritance

Public functions defines a *contract* with user...

- Derived class: do not add exception
- But we can remove some of them

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

class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException {};
  virtual void f() throw(DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw(BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw(BaseException) {
    throw BaseException();
  }
  virtual void g() throw(DerivedException) {
    throw DerivedException();
  }
};
```

Compiler : detects and signals errors

Inheritance

P

Compilation log

```
$ g++ -W -Wall -pedantic listings/Covariance.cpp -c -o listings/Covariance.o
listings/Covariance.cpp:18: error: looser throw specifier for 'virtual void Derived::f() throw (Base::BaseEx
listings/Covariance.cpp:8: error:   overriding 'virtual void Base::f() throw (Base::DerivedException)'
```

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

class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException {};
  virtual void f() throw(DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw(BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw(BaseException) {
    throw BaseException();
  }
  virtual void g() throw(DerivedException) {
    throw DerivedException();
  }
};
```

Compiler : detects and signals errors

Good practices

General rule

- If you are not sure of throwable exception then do not mention it.
- The stl don't specify any exceptions.

Example: standard stack: `std::stack`

```
T top() ;
void pop() ;
```

Why do we have 2 functions? $\rightarrow$ robust development.

Good practices

General rule

- If you are not sure of throwable exception then do not mention it.
- The stl don't specify any exceptions.

Example: standard stack: `std::stack`

```
T top() ;
void pop() ;
```

Why do we have 2 functions? → robust development. Software engineering principle:

- Each method makes ONE job.
- You must let the memory in the same state that before you (consistent memory/ no memory leaks).
- When you make affectation or equality test: do you want the structural one or the physical evaluation?

In brief: programming with exceptions

Don't use exception when

- you are programming asynchronous software ...
  - ... especially in handler!
- for insignificant error
  - handle them directly
- you want influence the data flow
  - The price is not free
- Exceptions are not required
  - make simple (with other feature if you want).
- Don't remodify old codes
  - if your code is fully correctly without why do you want lose time for that?

In brief: programming with exceptions

Use exceptions when

- Everywhere in specification : excepted in template ...
- Try to exploit existent exception (and override a text message`what()`)
- Avoid to declare in global namespace, do it as inner class or in a defined namespace
- Make a hierarchy ...
- Exploit multiple inheritance if needed
- Catch exception as reference!
- Be careful with constructor ...
- Never throw something in a destructor!
- Avoid *naked pointers*, give advantage *smart pointers*.