Exceptions

Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it.

- C++ provides exceptions to allow an error to be communicated.
- In C++ terminology, one portion of code throws an exception; another portion catches it.
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception.
- If an exception is not caught, the program terminates.

Throwing exceptions

- Exceptions in C++ are just normal values, matched by type.
- A class is often used to define a particular error type:
  ```cpp
  class MyError {};
  ```
- An instance of this can then be thrown, caught and possibly re-thrown:
  ```cpp
  void f() { ... throw MyError(); ... }
  ```

Conveying information

- The “thrown” type can carry information:
  ```cpp
  struct MyError {
    int errorcode;
    MyError(int i) : errorcode(i) {} }
  ```

  ```cpp
  void f() { ... throw MyError(5); ... }
  ```

  ```cpp
  try {
    f();
  } catch (MyError x) {
    // handle error (x.errorcode has the value 5)
    throw; // re-throw error
  }
  ```
Handling multiple errors

▶ Multiple catch blocks can be used to catch different errors:

```cpp
try {
...
}
catch (MyError x) {
  //handle MyError
}
catch (YourError x) {
  //handle YourError
}

▶ Every exception will be caught with `catch(...)`
▶ Class hierarchies can be used to express exceptions:

```cpp
#include <iostream>

struct SomeError {virtual void print() = 0;};

struct ThisError : public SomeError {
  virtual void print() {
    std::cout << "This Error" << std::endl;
  }
};

struct ThatError : public SomeError {
  virtual void print() {
    std::cout << "That Error" << std::endl;
  }
};

int main() {
  try { throw ThisError(); }
  catch (SomeError& e) { //reference, not value
    e.print();
  }
  return 0;
}
```

Exceptions and local variables

▶ When an exception is thrown, the stack is unwound
▶ The destructors of any local variables are called as this process continues
▶ Therefore it is good C++ design practise to wrap any locks, open file handles, heap memory etc., inside a stack-allocated class to ensure that the resources are released correctly

Templates

▶ Templates support *meta-programming*, where code can be evaluated at compile-time rather than run-time
▶ Templates support *generic programming* by allowing types to be parameters in a program
▶ Generic programming means we can write one set of algorithms and one set of data structures to work with objects of *any* type
▶ We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. sort routine presented earlier)
▶ The C++ Standard Template Library (STL) makes extensive use of templates
An example: a stack

- The stack data structure is a useful data abstraction concept for objects of many different types
- In one program, we might like to store a stack of \texttt{int}s
- In another, a stack of \texttt{NetworkHeader} objects
- Templates allow us to write a single \textit{generic} stack implementation for an unspecified type \texttt{T}
- What functionality would we like a stack to have?
  - \texttt{bool isEmpty()};
  - \texttt{void push(T item)};
  - \texttt{T pop()};
  - ...  
- Many of these operations depend on the type \texttt{T}

Creating a stack template

- A class template is defined as:

  ```c++
  template<class T> class Stack {
  ...
  }
  ```
- Where \texttt{class T} can be any C++ type (e.g. \texttt{int})
- When we wish to create an instance of a \texttt{Stack} (say to store \texttt{int}s) then we must specify the type of \texttt{T} in the declaration and definition of the object: \texttt{Stack<int> intstack};
- We can then use the object as normal: \texttt{intstack.push(3)};
- So, how do we implement \texttt{Stack}?
  - Write \texttt{T} whenever you would normally use a concrete type

```c++
1 template<class T> class Stack {
2  ...
3 }
4
5 #include "example16.hh"
6
7 template<class T> void Stack<T>::append(T val) {
8  Item **pp = &head;
9  while(*pp) {pp = &((*pp)->next);}
10  *pp = new Item(val);
11  }
12
13 //Complete these as an exercise
14 template<class T> void Stack<T>::push(T) {/* ... */}
15 template<class T> T Stack<T>::pop() {/* ... */}
16 template<class T> Stack<T>::~Stack() {/* ... */}
17
18 int main() {
19  Stack<char> s;
20  s.push('a'), s.append('b'), s.pop();
21 }
```

11 / 20 12 / 20
Template details

- A template parameter can take an integer value instead of a type:
  ```
  template<int i> class Buf { int b[i]; ... ; }
  ```
- A template can take several parameters:
  ```
  template<class T,int i> class Buf { T b[i]; ... ; }
  ```
- A template can even use one template parameter in the definition of a subsequent parameter:
  ```
  template<class T, T val> class A { ... ; }
  ```
- A templated class is not type checked until the template is instantiated:
  ```
  template<class T> class B {const static T a=3;};
  ```
  ```
  B<int> b;
  ```
  ```
  is fine, but what about
  ```
  B<B<int> > bi;
  ```
- Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Default parameters

- Template parameters may be given default values
  ```
  template <class T,int i=128> struct Buffer{
  T buf[i];
  };
  ```
- ```
  int main() {
  Buffer<int> B; //i=128
  Buffer<int,256> C;
  }
  ```

Specialization

- The class T template parameter will accept any type T
- We can define a specialization for a particular type as well:
  ```
  #include <iostream>
  class A {};
  ```
  ```
  template<class T> struct B {
  void print() { std::cout << "General" << std::endl;}
  }; 
  ```
  ```
  template<> struct B<A> {
  void print() { std::cout << "Special" << std::endl;}
  }
  ```
  ```
  int main() {
  B<A> b1;
  B<int> b2;
  b1.print(); //Special
  b2.print(); //General
  }
  ```

Templated functions

- A function definition can also be specified as a template; for example:
  ```
  template<class T> void sort(T a[],
  const unsigned int& len);
  ```
- The type of the template is inferred from the argument types:
  ```
  int a[] = {2,1,3}; sort(a,3); ⇒ T is an int
  ```
- The type can also be expressed explicitly:
  ```
  sort<int>(a)
  ```
- There is no such type inference for templated classes
- Using templates in this way enables:
  - better type checking than using `void *
  - potentially faster code (no function pointers)
  - larger binaries if `sort()` is used with data of many different types
```cpp
#include <iostream>

template<class T> void sort(T a[], const unsigned int& len) {
    T tmp;
    for(unsigned int i=0; i<len-1; i++)
        for(unsigned int j=0; j<len-1-i; j++)
            if (a[j] > a[j+1]) // type T must support "operator>"
                tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
}

int main() {
    const unsigned int len = 5;
    int a[len] = {1,4,3,2,5};
    float f[len] = {3.14,2.72,2.54,1.62,1.41};
    sort(a,len), sort(f,len);
    for(unsigned int i=0; i<len; i++)
        std::cout << a[i] << "t" << f[i] << std::endl;
}
```

---

### Overloading templated functions

- Templated functions can be overloaded with templated and non-templated functions
- Resolving an overloaded function call uses the "most specialised" function call
- If this is ambiguous, then an error is given, and the programmer must fix by:
  - being explicit with template parameters (e.g. `sort<int>(...)`)
  - re-writing definitions of overloaded functions
- Overloading templated functions enables meta-programming:

---

### Meta-programming example

```cpp
#include <iostream>

template<unsigned int N> inline long long int fact() {
    return N*fact<N-1>();
}

template<> inline long long int fact<0>() {
    return 1;
}

template<> inline long long int fact<0>() {
    return 1;
}

int main() {
    std::cout << fact<20>() << std::endl;
}
```

---

### Exercises

1. Provide an implementation for:
   - template<class T> T Stack<T>::pop(); and
   - template<class T> Stack<T>::~Stack();

2. Provide an implementation for:
   - Stack(const Stack& s); and
   - Stack& operator=(const Stack& s);

3. Using meta programming, write a templated class `prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.

4. How can you be sure that your implementation of class `prime` has been evaluated at compile time?