LECTURE 3

Template functions

A simple function returning the max value of two integers:

```
1 int myMax(int a, int b)
2 {
3   return (a<b ? b : a);
4 }
```

- What if one wants to have a similar function for other types (e.g. `double`'s, or any type with a supported operator <)?
- It's possible to define functions with the same name but different argument types. However apart from the type names, the code would be the same.
- Note: code duplication is always risky. (Discussion question: Why?)
- Another solution: **template function**:

```
1 template<typename T>
2 T myMax(T a, T b)
3 {
4   return (a<b ? b : a);
5 }
```

Template functions explained

```
1 template<typename T>
2 T myMax(T a, T b)
3 {
4   return (a<b ? b : a);
5 }
```

- In the above function the type `T` is a dummy type, it is referred to as **template parameter**.
- This piece of code is not a function (yet).
- It's technically just instructions to the compiler: if the function `myMax<arg_type>` is called with a particular (non-dummy) type `arg_type`, then take that type and substitute into the place of the dummy type `T` and generate a particular function.
- If the function is called with another type, the compiler will generate another version of the function.
- If the arguments of the function uniquely determine the template type argument (see examples), then there is no need to specify the template parameter at call.
- The keyword `typename` is used before the template parameter, one can use the keyword `class` instead.
- Multiple template parameters are allowed.
- If the function is called with a template parameter for which the operator < is not defined, the compiler will generate an error. (Debug problems.)

Calling template functions

```
1   int a(1), b(2);
2   cout << "The max of " << a << " and " << b << " is: "
3    << myMax(a,b) << endl;
```

In this case, no specification of template parameter is required, the function arguments determine that.

```
1   int a(1);
2   double c(1.25);
3   cout << "The max of " << a << " and " << c << " is: "
4    << myMax(a,c) << endl;
```

Compiling error: "template parameter 'T' is ambiguous".

```
1   int a(1);
2   double c(1.25);
3   cout << "The max of " << a << " and " << c << " is: "
4    << myMax<double>(a,c) << endl;
5   cout << "The max of " << a << " and " << c << " is: "
6    << myMax<int>(a,c) << endl;
```

One can use either `double` or `int` as template parameter, in both cases the type of one of the arguments will be converted, with possible loss of data.

Calling template functions

```
1   int a(1);
2   double c(1.25);
3   cout << "The max of " << a << " and " << c << " is: "
4    << myMax<double>(a,c) << endl;
5   cout << "The max of " << a << " and " << c << " is: "
6    << myMax<int>(a,c) << endl;
```

```
The max of 1 and 1.25 is: 1.25
The max of 1 and 1.25 is: 1
```

Note the side effect of type conversion.

Template `myComparison` with special ordering

```
1  template<typename T, typename op>
2  T myComparison(T a, T b){
3    return (op()(a,b)? b : a);
4  }
```

- What if one would like to have a bit more general max function with special ordering rule?
- In the above example, the template parameters `T` and `op` are assumed to provide the following properties:
  - `op` is a function object class with operator `()` implemented, taking two arguments both of type `T` and returning a `bool`,
  - furthermore, the `op` is assumed to have a default constructor.
- Note that an instance of `op` is constructed and its operator `()` is called (in the function body).
- Discussion question: Why is this implementation not efficient?
- Improvement: `static` instance of `op`

Template classes - a function object example

```
1  template<typename T, typename Op=std::less<T> >
2  class myComparisonFunObj
3  {
4  public:
5    myComparisonFunObj(Op opArg=Op()) : m_Op(opArg) {}
6    T operator()(const T & a, const T & b){
7      return (m_Op(a,b)? b : a);
8    }
9  private:
10   Op m_Op;
11 };
```

- Similarly to functions, in C++ one can implement **template classes**.
- New features: default template parameter can be defined, non-type (but value) template parameters allowed, (partial) template specialization is available (see later).
- This class holds an operator of dummy type `Op` as a member:
  - `Op` assumed to have an operator `()` implemented taking two arguments of type `const T` by ref and returning a `bool`.
  - furthermore `Op` is assumed to have a default and a copy constructor implemented.
- If the template parameter `Op` is not specified, `less<T>` is taken by default as type `Op`.
- If no instance of `Op` is passed to the constructor, the default constructed `Op` is used.

Template classes in action

```
1   myComparisonFunObj<int> myMax;
2   myComparisonFunObj<int,std::greater<int> > myMin;
3   cout << "Using op=less, " << a << " and " << b << " is: "
4    << myMax(a,b) << endl;
5   cout << "Using op=greater, " << a << " and " << b << " is: "
6    << myMin(a,b) << endl;
```

```
Using op=less, 1 and 2 is: 2
Using op=greater, 1 and 2 is: 1
```

- MyMax is of type myComparisonFunObj<int,less<int> > (note the space between > >), where the second template parameter is determined implicitly: the default one: less<int> is used.
- MyMin is of type myComparisonFunObj<int,greater<int> >, the second template parameter is specified explicitly.
- Note: **The type of an instance of a template class is determined by the name of the class and the realized template parameters**,
- i.e. MyMax and MyMin are not of the same type, they cannot be stored in the same stl container (except in a pair or a container of pairs).

Templates with non-type arguments and specialization

```
1   template<unsigned int n>
2   class Fibonacci
3   {
4   public:
5     static const int result =
6       Fibonacci<n-1>::result+Fibonacci<n-2>::result;
7   };
8   //specialization of Fibonacci<n> for n=0
9   template<>
10  class Fibonacci<0>
11  {
12  public:
13    static const int result=1;
14  };
15  //specialization of Fibonacci<n> for n=1
16  template<>
17  class Fibonacci<1>
18  {
19  public:
20    static const int result=1;
21  };
22     //***************************************
23     //Fibonacci numbers (e.g. in main):
24     int fib10=Fibonacci<10>::result;
```

Templates with non-type arguments and specialization

```
1  template<unsigned int n>
2  class Fibonacci
3  {
4  public:
5    static const int result =
6      Fibonacci<n-1>::result+Fibonacci<n-2>::result;
7  };
```

- class definition for a general integer `n`
- for each particular value of `n`, the compile implements a different class
- the type of the class depends on the template parameter `n`

A note on the `static const` member

- `static` data members are shared by all instances of the class
- `static` data members are initialised and accessible even if no instance of the class has been constructed
- therefore, static data members must be explicitly initialise - typically outside the class body, except in some special cases
- note: `int fib10=Fibonacci<10>::result;` - no instance of `Fibonacci<10>` is created, the shared `static` member is accessed

Templates with non-type arguments and specialization

```
1  //specialization of Fibonacci<n> for n=0
2  template<>
3  class Fibonacci<0>
4  {
5  public:
6    static const int result=1;
7  };
8  //specialization of Fibonacci<n> for n=1
9  template<>
10 class Fibonacci<1>
11 {
12 public:
13   static const int result=1;
14 };
```

- when `Fibonacci<n>` or its member is to be accessed, the compiler will generate the code for the class `Fibonacci<n>`
- since the implementation of `Fibonacci<n>` requires the class `Fibonacci<n-1>` if `n>` 1, the compiler will recursively generate code for the classes `Fibonacci<i>` for `i=` $n-1, n-2, \ldots, 0$
- the template specialisation in this particular case defines the initial value for the sequence
- the compiler generates the code and the values of each of the `static` members
- the compiler has to know the value of `n` **at compile time**, otherwise cannot generate the code - `n` cannot be specified during run time

Templates with non-type arguments

Non-type template parameters

- Values of the following types are allowed as non-type template parameters: integral, enumeration, pointer, reference, or pointer to member type, and **must be constant at compile time**.
- Floating point values, objects of class, struct are not allowed as template parameters.
- The specialization is generated by the compiler, in particular `Fibonacci<10>::result` is worked out by the compiler.

Templates specialization

```
1  template<typename A,typename B>
2  class ClassName
3  {//general implementation...};
4
5  //partial specialization: B is set to int
6  template<typename A>
7  class ClassName<A,int>
8  {//specialized implementation...};
9
10 //full specialization: A is string, B is int
11 template<>
12 class ClassName<string,int>
13 {//specialized implementation...};
```

- the general behaviour for general types `A` and `B` is defined first
- partial specialisation: the behaviour might be different if one of the type parameters is something specific, e.g. `int`
- in case of partial specialisation the non specialised type arguments must be listed: `template<typename A>`
- full specialisation: the behaviour might be very specific for a particular template argument list
- in case of full specialisation an empty template argument list must be provided `template<>`

Templates - general remarks

- By using template functions and template classes, one can save time and eliminate bugs due to code duplication.
- By using templates, usually certain assumptions are made about template parameters, these assumptions are sometimes implicit and hard to debug errors due to them.
- Template functions and classes are compiled only if used/defined. I.e. if no instances are defined, no compiler error is generated, which does not imply the correctness of the code!
- The header-cpp file structure does not work for templates the usual way. The simplest solution is to keep the template implementations in the header files.
- Template classes are often used for implementing **static polymorphism**, types are fixed at compile time. (Recall, the type of the template class is determined by its name and the particular template parameters (types or value)).
- Templates are generated at compile time, i.e. some of the computations may be redirected to the compiler, improving the runtime performance.
- **Template metaprogramming** is yet another coding approach based on templates. It's about making the compiler to "work out things which otherwise would be very hard or even impossible" (ref. and more details: *Template metaprogramming* by Abrahams & Gurtovoy, Addison-Wesley 2005).

Summary

template functions	`typename`
template classes	function objects
`static const` members	non-type template arguments
partial specialisation	full specialisation

Linked list

A (singly) linked list consists of **nodes**. Each node:

- has some data (an `int` in our case),
- has **set** and **get** methods for this data,
- knows about the **next** (but only the next) node,
- can tell **if it has** a next node,
- has a **get** method for the next node.

Moreover, it is possible to **append** a new node to the **end** of the list.

First attempt at implementing a linked list

```
1  class LinkedList01
2  {
3  public:
4    LinkedList01(int);
5
6    LinkedList01 * getNext() const;
7    void append(int);
8    bool hasNext() const;
9    int getData() const;
10   void setData(int);
11
12   std::ostream & print(std::ostream &) const;
13
14 private:
15   int mData;
16   LinkedList01 * mNext;
17 };
```

```
1    int mData;
2    LinkedList01 * mNext;
```

- the data of a node is stored in a data member `mData`,
- the address of the next node is stored in `mNext`.

```
1  LinkedList01::LinkedList01(int arg) : mData(arg) {
2    mNext = NULL;
3  }
```

- the constructor sets the data member,
- and sets the next to `NULL`, that is no memory is allocated yet, (see `nullptr` in C++11).

```
1 int LinkedList01::getData() const {
2   return mData;
3 }
```

- `getData()` returns the **value** of the data member.

```
1 void LinkedList01::setData(int arg){
2   mData = arg;
3 }
```

- `setData()` sets the value of the data member,
- the function can validate the input, and reject the modification.

Member implementations

```
1 bool LinkedList01::hasNext() const {
2   return bool(mNext);
3 }
```

- `hasNext()` checks if the pointer is not `NULL` – note the conversion to `bool`.

```
1 LinkedList01 * LinkedList01::getNext() const {
2   return mNext;
3 }
```

- `getNext()` returns the address of the next node (no type conversion).

```
1 void LinkedList01::append(int arg) {
2   if(hasNext())
3     mNext->append(arg); // note: recursive fn call
4   else
5     mNext = new LinkedList01(arg);
6 }
```

- `append()` creates a new node at the **end** of the list with the given data. Note: the new node is created on the **heap**.

First attempt – what can go wrong?

Test function:

```
template<class L>
void testList(unsigned int m, unsigned int n) {
  for(unsigned int i=0; i<m; i++){
    // create list
    L l(0);
    if(i % 100 == 0)
      cout << "\ti: " << i << endl;
    for(unsigned int j=1; j<n; j++)
      l.append(j);
    // local l is destroyed
  }
}
```

- Template function: one function to test all list versions (details will be given later),
- creates `m` lists, each of length `n`,
- note the scope: each list is local, and the destructor (where is the destructor?) is called before the next list is created.

Observation: memory constantly increases as the function is executed; the nodes are not cleaned up – **memory leak**.

Side note – running code with arguments

```
int main(int argc, char* argv[]) {
  if(argc<4){
    cout << "At least three arguments are expected" << endl;
    return 0;
  }

  unsigned int m(atoi(argv[1]));
  unsigned int n(atoi(argv[2]));
  unsigned int t(atoi(argv[3]));
```

- `main()` can take arguments: a number, and an array of `char` arrays (i.e. C-style strings),
- the number specifies the number of arguments (the length of the array of strings) taken by the executable, (note: the first string in the array is always the name of the executable),
- on linux and mac, the executable can be run as

```
>> ./Lecture06 50 500 1
```

- in Visual Studio, set these as command line arguments,
- the string arguments are to be converted to the right type.

Second attempt at implementing a linked list

What if we make sure that the memory is released when the nodes are destroyed?

```
1  class LinkedList02
2  {
3  public:
4    LinkedList02(int);
5    ~LinkedList02();
6
7    LinkedList02 * getNext() const;
8    void append(int);
9    bool hasNext() const;
10   int getData() const;
11   void setData(int);
12
13   std::ostream & print(std::ostream &) const;
14
15 private:
16   int mData;
17   LinkedList02 * mNext;
18 };
```

Second attempt at implementing a linked list

```
1  LinkedList02::~LinkedList02(){
2    if(hasNext())
3      delete mNext;
4  }
```

- A custom destructor is declared and implemented,
- the destructor calls `delete` on the node.
- In particular, when the head node is destroyed, all its subsequent nodes are destroyed, and the memory is released.

Second attempt – what can go wrong?

```
1  template<class L>
2  void dummyPrint(L l) {
3    cout << "List printed from dummyPrint: " << l << endl;
4  }
5
6  template<class L>
7  void testDummyPrint() {
8    L l(123);
9    l.append(234);
10   l.append(345);
11   cout << "Testing pass-by-value" << endl;
12   cout << "Test list: " << l << endl;
13   dummyPrint(l);
14   cout << "Test list after dummy print: " << l << endl;
15 }
```

- Template function: one function to test all list versions (details will be given later),
- `testDummyPrint()` creates a list, then calls `dummyPrint()` on this list,
- `dummyPrint()` takes its argument **by value**, hence creates a copy using the copy constructor (where is the copy constructor??),
- the copied list is destroyed at the end of the scope of `dummyPrint()`,
- but: the head of the copied list points to the same node as the original! Therefore, all the subsequent nodes in the original list are deleted.

Second attempt – what can go wrong?

How to resolve this issue?

- Don't pass the list by value, pass it **by reference** instead. (OK, but copies of lists may still get created accidentally, hence the destructor should properly do its job.)
- Don't let the copy and the original list share any nodes **–** implement a copy constructor that creates a **deep copy** of the original. (OK, deep copies are often useful, and safer than shallow copies, but what if we still need **shallow** copies for various reasons?)
- Somehow, let the (shallow) copy know about the original list still wanting to access its nodes. But careful:
  - there might be multiple shallow copies floating around,
  - the original list may get deleted before some of the shallow copies (how?), in which case the original list should not delete its nodes.

Resources are created, get shared and destroyed in the following members:

- **constructor**,
- **copy constructor**,
- **copy assignment**,
- **destructor**.

We have to carefully (re)implement these.

Third attempt at implementing a linked list

How to keep track of lists that point to a given node?

```
class LinkedList03
{
public:
  LinkedList03(int);
  LinkedList03(const LinkedList03 &);
  ~LinkedList03();
  LinkedList03 & operator=(const LinkedList03 &);
  LinkedList03 * getNext() const;
  void append(int);
  bool hasNext() const;
  int getData() const;
  void setData(int);
  std::ostream & print(std::ostream &) const;

private:
  void releaseNext();
  void acquireNext(LinkedList03 *);
private:
  int mData;
  LinkedList03 * mNext;
  unsigned int mRefCount;
};
```

Third attempt at implementing a linked list

Three `private` members have been added:

```
  void releaseNext();
  void acquireNext(LinkedList03 *);
  unsigned int mRefCount;
```

- `mRefCount` is a member to store the **number of owners**.

```
void LinkedList03::acquireNext(LinkedList03 * arg){
  mNext = arg;
  if(arg)
    arg->mRefCount++;
}
```

- When (re)assigning the next node, `acquireNext()` increments the counter of the next by one.

```
void LinkedList03::releaseNext(){
  if(hasNext() && (mNext->mRefCount > 1))
    mNext->mRefCount--;
  else if(hasNext())
    delete mNext;
}
```

- `releaseNext()` deletes the next node only if the node's counter is not greater than 1 (i.e. the list that is being destructed is the only owner),
- Otherwise, the node is not deleted, but only its counter is decreased by 1.

Third attempt at implementing a linked list

```
1 LinkedList03::LinkedList03(int arg) : mData(arg), mRefCount(0) {
2   mNext = NULL;
3 }
```

- The **constructor** initialises `mData` and sets the counter to 0.

```
1 LinkedList03::LinkedList03(const LinkedList03 & arg)
2 : mData(arg.mData), mRefCount(0) {
3   acquireNext(arg.mNext);
4 }
```

- The **copy constructor** initialises `mData` and sets the counter to 0.
- Moreover, it acquires the next node; in particular, the next node gets an additional owner.

```
1 LinkedList03::~LinkedList03(){
2   releaseNext();
3 }
```

- The **destructor** releases the next node; decrements the next node's counter or deletes if it has no more owners.

Third attempt at implementing a linked list

```
1 LinkedList03 & LinkedList03::operator=(const LinkedList03 & arg){
2   releaseNext();
3   mData = arg.mData;
4   acquireNext(arg.mNext);
5   return *this;
6 }
```

- The **copy assignment** copies `mData`, leaves the counter as it is.
- Moreover, the pointer to the next node is to be updated: the current next node is to be released first, and then the new next node can be acquired.

```
1 void LinkedList03::append(int arg) {
2   if(hasNext())
3     mNext->append(arg); // note: recursive fn call
4   else
5     acquireNext(new LinkedList03(arg));
6 }
```

- The **append** member instead of just assigning a new node to `mNext` properly acquires a new next node.

Third attempt – anything can go wrong?

Yes, unfortunately . . . this solution requires a bit more work to be **thread safe**. . . more care is required when checking and modifying the reference counters.

Fourth attempt at implementing a linked list

Similar effect, with less work, and with thread safety.

```
1  class LinkedList04
2  {
3  public:
4    typedef ptrnamespace::shared_ptr<LinkedList04> LinkedList04Ptr;
5
6    LinkedList04(int);
7
8    LinkedList04Ptr getNext() const;
9    void append(int);
10   bool hasNext() const;
11   int getData() const;
12   void setData(int);
13
14   std::ostream & print(std::ostream &) const;
15
16 private:
17   int mData;
18   LinkedList04Ptr mNext;
19 };
```

The implementation is very similar to the first one, except for

```
typedef ptrnamespace::shared_ptr<LinkedList04> LinkedList04Ptr;
```

- `shared_ptr<LinkedList04>` is used instead of a raw pointer (from `boost` or `std` of C++11),
- it has built-in thread safe reference counter,
- it properly cleans up the memory, etc.

Fourth attempt – surely, nothing can go wrong, right?

Unfortunately, . . .

- it depends on the implementation of the compiler,
- the destruction of a long list is a recursive call to destructors,
- if not optimised (it is not in the MS compiler), the stack may overflow,
- increasing the stack size only postpones the occurrence of the problem.

Summary

pointer	heap
`new`	`delete`
pointer member	memory leak
reference counting	shared pointer
(copy) constructor	destructor