UNIT-5

Generic programing with templates and Exceptional handling

Generic programming with

* Template provides generic programming by defining generic classes.

* Template is a technique are passes (or) keyword that allows a using a single function (or) class supports different types of data

* The Goal of generic programming is write the code that is independend of datatype.

* A function that works for all C++ datatypes is called as generic functions.

* Templates associated with classes are called as template classes

Ex:- * If we can create a template for addition of two numbers it helps us to calculate addition of any datatype including int, float, class, double, long, small

* Using template we can create a single function that can process any type of data i.e the formal parameters of functions are template types

original data with some value
actual parameters
formal

* Normally we overload a function when we need to handling different data types this approach increases size of the program and local variables are created in the memory

Defination (or) Declaration of Template:-

To declare a template class following syntax is used

```
template calls <t>
class class_name
{
data member declaration,
member function declaration/definition
};
```

* the first statement in the above syntax also the compiler that the following class is gives template datatype
* t is a variable of template datatype that can be used in following class
* < > it is the angular brackets used to declare template variables.

i) Write a program to show values of different datatypes using constructor overloading and templates

Constructor overloading

```
# include <iostream.h>
# include <conio.h>
class data
{
public:
  data (char c)
  {
  cout << "\n" << "c = " << c << " size in bytes:" << sizeof [c];
  }
  data (int c)
  {
  cout<<"\n"<< "c = "<<c<< "size of in bytes:"sizeof [c];
  }
 data (double)
 {
 cout << "\n" <<"c=" <<c<<"size in bytes:"<< c sizeof [c];
 }
};
```

```
void main()
{
clrscr();
data h('A');
data i(100);
data j(68.2);
getch();
}
```

O/P

C = A
Size in bytes = 1
C = 100
Size in bytes = 2
C = 68.2
Size in bytes = 8

Template:-

```
# include<iostream.h>
#l include<conio.h>
Template <class T>
Class data
{
 public;
  data (T c)
 {
cout <<"\n"<<"c="<<c<<" Size in bytes:"<<sizeof (c);
 }
};
void main()
{
 clrscr();
 data <char> h('A');
 data <int> i(100);
 data <float> j (3.12);
 getch();
}
```

O/P

C=A
Size in bytes=1
C=100
Size in bytes =2
C=68.2
Size in bytes =8

Normal function Templates (or) Function templates:-

* A normal function can also used template arguments.
* The difference between normal and member function is normal member function (or) defined without classes that is they are not members of any class member functions are defined by using classes that is they can be invoked by using object and DOT operator.

write a program to define a normal template function

```
#include<iostream.h>
#include<conio.h>
 template <class T>
   void show(T x)
   {
    cout<<"x value:"<<x;
   }
   void main()
    {
      clrscr();
       int i=100;
       char c='A';
       float f=68.50;
       show(i);
       show(c);
       show(f);
       getch();
       }
```

output:-

x value=100
x value= A
x value=68.50

2) write a program to define data members of template type.

```
#include <iostream.h>
#include <conio.h>
template <class t>
class data
{
public:
  data(t x)
  {
   cout <<"x value:" <<x;
  }
};
void main()
{
 clrscr();
 data <int> i(100);
 data <char> c(A);
 data <float> f(68.50);
 getch();
}
```

Output

x value =100;

x value = A;

x value = 68.50;

write a program find out square of the given value by using template

```
#include<iostream.h>
#include<conio.h>
template<class T>
class square
{
public:
    square(T x)
    {
    cout<<"x value:"<<x*x;
    }
};
void main()
{
clrscr();
square<int>i(10);
square<float>f(6.5);
getch();
}
```

2) write a c++ program for swapping two values using function templates

```
main()
{
int a,b;
cout<<" enter a,b values:";
cin>>a>>b;
a=a+b;
b=a-b;
a=a-b;
cout<<" after swapping "<<a<<b;
}
```

without using templates

```
#include<iostream.h>
void swap(int *a, int *b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}
void main()
{
 int x,y;
 clrscr();
 cout<<" enter any two integers";
 cin>>x>>y;
 cout<<"Before swapping two numbers are:"<<x<<y;
  swap(&x, &y);
 cout<<"after swapping two numbers are:"<<x<<y;
 getch();
}
```

Using template :-

```
#include <iostream.h>
#include <conio.h>
template <class T>
void swap(T *a, T *b)
{
 T temp;
 temp = a;
 a = b;
  b = temp;
 }
void main()
{
  int x,y
  float c,d;
  clrscr();
   cout <<" Enter any two integers values";
   cin >>x>>y
cout<<" Before swaping two numbers are ="<<x<<y;
  swap (&x, &y);
cout<<" after swaping two numbers are ="<<x<<y;
 cout<< "Enter any two float values";
  cin>>c>>d;
 cout<<" Before swapping two float values are"<<c<<d;
  swap(&c,&d)
 cout <<" after swapping two float values are"<< c<<d;
  getch();
 }
```

2 Write a program to find out maximum of two numbers by using templates.

```
#include <iostream.h>
#include <conio.h>
template <class T>
void max(T a, T b)
{
  if (a>b)
  cout << "a is maximum number";
  else
  cout << "b is maximum number";
}
void main()
{
  int a,b;
  float c,d;
  clrscr();
  cout << "read a,b integer values";
  cin >> a >> b;
    max(a,b);
  cout << "read c,d float values";
  cin >> c >> d;
    max(c,d);
  getch();
}
```

Output:- Read a,b Integer values 10 12

b is maximum number

Read c,d float values 12.50 10.50

a is maximum number

write a c++ program to illustrate -template class

```
#include <iostream.h>
#include <conio.h>
template <class T1,T2>
  class test
  {
  public:
  T1 a;
  T2 b;
  test (T1 x, T2 y)
   {
     a=x;
     b=y;
   }
   void show( )
   {
   cout <<"a=" <<a<<"\n";
   cout <<"b=" <<b;
   }
};
void main( )
{
 clrscr( );
 test <int, float> t1(10, 20.50);
 test <float, int> t2(20.50,10);
  t1.show( );
  t2.show( );
  getch( );
}
```

output:

a = 10

b = 20.50

a = 20.50

b = 10

Overloading of template functions:-

* A template function supports overloading mechanism it can be overloaded by normal function (or) template function.
* The compiler follows the following rules for choosing appropriate function when program consist of template and normal function

i) search for accurate match of functions if found it is invoked

ii) In case no match is found on error will be reported

Write a program to overload template function

```
#include <iostream.h>
#include <conio.h>
template <class t>
void show(t a)
{
 cout <<" template variable =" << a;
}
void show(int a)
{
cout <<" integer variable =" <<a;
}
void main()
{
 clrscr();
 show('A');
 show(100);
 show(10.50);
 getch();
}
```

O/P:-

template variable = A

Integer variable : 100

template variable = 10.50

Bubble sort using Function template:-

* The process of arranging the giving elements either in ascending (or) descending order is called as sorting.

* Bubble sort is one of the technique for sorting given elements

* In bubble sort we are comparing adjacent elements

ex:- apply bubble sort technique for following data and place the data in ascending order.

12, 3, 0, -3, 1, -9

* place the given elements in array

a[5] =

0	1	2	3	4	5
12	3	0	-3	1	-9

Step Step: 1

Compare a[0] and a[1]

Compare 12 and 3

12>3

Swapping two elements

0	1	2	3	4	5
3	12	0	-3	1	-9

Compare a[1] and a[2]

Compare 12 and 0

12>0

Swapping two elements

0	1	2	3	4	5
3	0	12	-3	1	-9

Compare a[2] and a[3]

Compare 12 and -3

12>-3

Swapping two elem

0	1	2	3	4	5
3	0	-3	12	1	-9

Compare a[3] and a[4]

Compare 12 and 1

12>1

Swapping two elements

0	1	2	3	4	5
3	0	-3	1	12	-9

Compare a[4] and a[5]

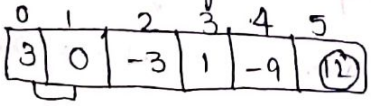
Compare 12 and -9

12>-9

0	1	2	3	4	5
3	0	-3	1	-9	12

* Therefore in step one we placing (or) sorting the largest element in given array.

Step:2 a[5] =



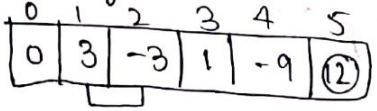
Step:2

Compare a[0] and a[1]

Compare 3 and 0

3>0

Swapping two elements

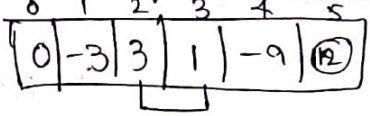


Compare a[1] and a[2]

Compare 3 and -3

3 > -3

Swapping two elements

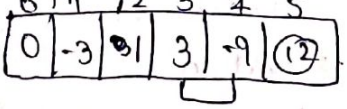


Compare a[2] and a[3]

Compare 3 and 1.

3>1

Swapping two elements

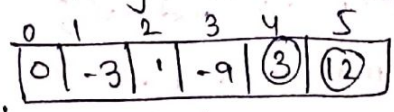


Compare a[3] and a[4]

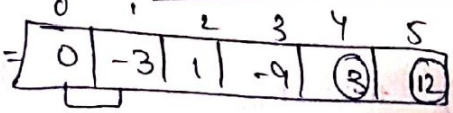
Compare 3 and -9

3>-9

Swapping two elements



a[5] =

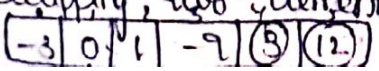


Step:3:- Compare a[0] and a[1]

Compare 0 and -3

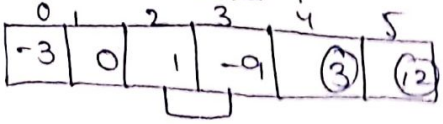
0>-3

Swapping two elements



Compare a[1] and a[2]

Compare 0 and 1

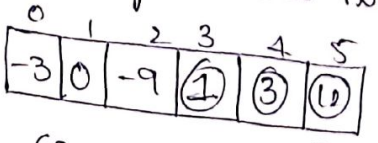


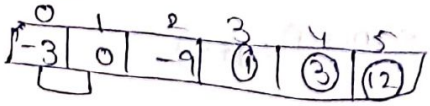
Compare a[2] and a[3]

compare 1 and -9

1 > -9

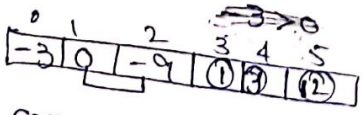
Swapping of two numbers



Now a[5] = 

Step:4 Compare a[0] and a[1]

Compare -3 and 0.

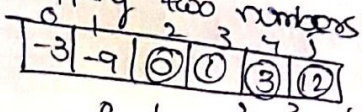


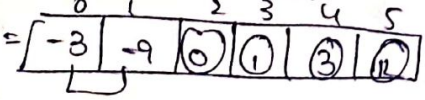
Compare a[1] and a[2]

Compare 0 and -9

0 > -9

Swapping two numbers



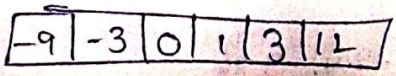
a[5] = 

Step:5 Compare a[0] and a[1]

Compare 0 and 1

-3 > -9

Swapping two elements



**

Write a program sort the given elements by using bubble, using template function.

```
#include<iostream.h>
#include<conio.h>
template<class T>
void Bsort(T a[], int n)
{
 for(int i=0; i<n-1; i++)
  {
  for(int j=0; j<(n-i-1); j++)
   {
     if(a[j]>a[j+1])
     {
       T temp;
      temp=a[j];
       a[j]=a[j+1];
      a[j+1]=temp;
     }
   }
  }
}
void main()
{
 int i;
 clrscr();
 int a[5]={12,0,3,1,-9,-3}
 float b[5]={12.5,0.4,3.2,1.6,-9.4,-3.8}
  Bsort(a,6); - function calling statement
  Bsort(b,6);
 cout<<" after sorting integer elements are";
   for(i=0; i<6; i++)
   cout<<a[i]<<"\t";
```

```
cout <<"after sorting float elements";
for (i=0; i<6; i++)
  cout <<b[i]<<"\t";
}getch();
```

Output :- after sorting integer elements are

-9, -3, 0, 1, 3, 12

after sorting float elements are

-9.4, -3.8, 0.4, 1.6, 3.2, 12.5

Difference between Template and Macros :- (preprocessing)

* Macro's are preprocessing statements and not type safe that is a macro defined for integer operation can't accept the float data.
* It is difficult to find out errors in macros.

Write a program to perform increment operation by using Macros and templates.

```
#include <iostream.h>
#include <conio.h>
#define mac(x) ++x
void main()
{
 int a,c;
 clrscr();
 a=10;
 c=mac(a);
 cout<<"after incrementation of a="<<c;
 getch();
}
```

```
#include <iostream.h>
#include <conio.h>
template <class T>
T mac(T x);
{
 ++x;
 return x;
}
void main()
{
 int a,c;
 clrscr();
 a=10;
 c=mac(a);
 cout<<"after incrementation of
 getch();          a="<<c;
}
```

Exceptional handling:-

* Developing an error free program is the objective of programer. the programer have to take care to prevent errors. Errors can be handled by using exception handling Mechanism.

* the errors may be syntax errors (or) logical errors

* the logical errors occured in the program due to poor understanding of the program (logic)

* Syntax errors occured dee to in the program due to poor understanding of the programing languages

* C++ Provides exceptional handling procedure to reduce the errors that a programer makes

* programer always faces unusual errors while writting programs

* An exception is abnormal termination of a program which is executed in a program at run time.

Principles of exception Handling:-

* the goal of exception Handling is to create a routine that identifies and stops an exceptionals inorder to execute program properly.

* the routine needs to carry the following responsibilities

i) Identify an exception (or) Identify an problem

ii) Warn that an error has come

iii) Accept the error message

iv) Handling the error message

v) An exception is an object in his shift from path of the program where an error occurs to that path of

the program and which is going to handle the error.

Keywords of exceptional handling:-

* Exception handling technique passes control of the program from a location of exception in a program to an exception handler with the try block.

i) Try Block:- * the try keyword is followed by a series of statements enclosed with curly braces.

* Try keyword is used to identifying an exceptions

ii) throw keyword (or) Block:- * the function of throw statements are to send the exception found.

* the declaration of throw statement is as given below.

```
throw (exception);
  (or)
throw exception;
```

* The argument exception is allowed in any type of data and it may be a constant value also

iii) catch Block (or) keyword:- * like try blocks catch block also contains a series of statements enclosed within curly brass.

* catch block associated with try block catches the exceptions thrown and the control is transfered from try block to catch block.

Syntax:-

```
try
{
  Statement 1;
  Statement 2;
  :
  :
  statement n;
}
catch(type z argument)
{
 Statement 1;
 :
 statement n; }
```

1) write a program to identify exceptions by using exception handling mechanism.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a,b,x;
clrscr();
cout<<"Enter any two numbers";
cin>>a>>b;
x=a-b;
try
{
if(x!=0)
cout<<"Result="<<b/x;
else
throw(x);
}
}
catch(int i)
{
cout<<"exception caught:"<<i;
}
```

output:-

Enter any two numbers 20 20

Result = 1

Enter any two numbers 10 10

exception caught : 0

Multiple Catch statements:- It is possible to a program segment has more than one catch statement with a single try block.

Syntax:-

```
try
{
 // try block
}
catch (Type 1 argument)
{
  // catch Block
}
Catch (Type 2 argument)
{
 // catch block
 }
 :
Catch (type N argument)
{
 //catch Block
}
```

* when an exception is thrown the exception handler are searched in order for an appropriate match

write a program to catch multiple exceptions.

```
#include <iostream.h>
#include <conio.h>
void num(int k)
{
 try
 {
 if(k==0) throw k;
  else
   if(k>0) throw 'p';
  else
   if (k<0) throw '0';
  cout<<"*** try block ***\n";
 }
  catch (int i)
```

```
{
cout<<"\n caught an exception \n";
}
}catch(char ch)
void main(-)
{
cout<<"\n caught an exception \n"<<ch;
}
}

void main()
{
 num(0);
 num(5);
 num(-1);
 getch();
}
```

output:- caugh an exception: 0

caught

catch in multiple exceptions:-

*It is also possible to define a single catch block from one (or) more exceptions of different datatypes. In such situation single catch block is used the catch exceptions thrown by multiple throw statements

Syntax:-

```
catch(.....)
{
  //catch block statements
}
```

write a program to handle multiple exceptions by using a single catch block.

```
# include <iostream.h>
# include <conio.h>
void num(int k)
{
 try
 {
```

```
    if(k==0) throw k;
    else
    if(k>0)throw 'p';
    else
     if (k<0) throw 0.5;
    }
}
catch (....)
 {
 cout<<"exception caught";
 }
void main()
 {
  clscr();
  num(0);
  num(-1);
  num(1);
  getch();
 }
```

advantage :- reduce the size of the program.

disadvantage :- we don't find what kind of exception

output:-

exception caught

exception caught

exception caught

Specifying exceptions:-

The specified exceptions are used when we want to tell bind the function to throw only specified exceptions using a throw list condition can do this.

Syntax:-

```
Returntype functionname(parameters list) throw (type of exception)
 {
   function Body
 }
```

write a program to handle only integer type of exceptions.

```
#include<iostream.h>
#include <conio.h>
void num(int k) throw(int)
 {
  try
    {
      if(k==0) throw k;
      else if(k<0) throw k;
      else
         throw 0.5;
    }
  }
 catch (int i)
 {
  cout<<"Integer exception caught"<<i;
 }
 catch (char ch)
 {
  cout<<" character exception caught"<<ch;
 }
 catch(float f)
 {
 cout<<"float exception caught"<< f;
 }
 void main()
 {
  clrscr();
   num(0);
   num(-1);
  num(1);
   getch();
 }
```

output:-

Integer exception caught 0

C and C++

7. Exceptions — Templates

Stephen Clark

University of Cambridge
(heavily based on last year's notes (Andrew Moore) with thanks to Alastair R. Beresford and Bjarne Stroustrup)

Michaelmas Term 2011

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:
  `class MyError {};`
- An instance of this can then be thrown, caught and possibly re-thrown:

```
1 void f() { ... throw MyError(); ... }
2 ...
3 try {
4   f();
5 }
6 catch (MyError) {
7   //handle error
8   throw; //re-throw error
9 }
```


Conveying information

- The "thrown" type can carry information:

```
1  struct MyError {
2    int errorcode;
3    MyError(i):errorcode(i) {}
4  };
5
6  void f() { ... throw MyError(5); ... }
7
8  try {
9    f();
10 }
11 catch (MyError x) {
12   //handle error (x.errorcode has the value 5)
13   ...
14 }
```

Handling multiple errors

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

```
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 `int`s
- In another, a stack of `NetworkHeader` objects
- Templates allow us to write a single generic stack implementation for an unspecified type `T`
- What functionality would we like a stack to have?
  - `bool isEmpty();`
  - `void push(T item);`
  - `T pop();`
  - ...
- Many of these operations depend on the type `T`

Creating a stack template

- A class template is defined as:

```
1 template<class T> class Stack {
2   ...
3 }
```

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


```
template<class T> class Stack {

  struct Item { //class with all public members
    T val;
    Item* next;
    Item(T v) : val(v), next(0) {}
  };

  Item* head;

  Stack(const Stack& s) {}              //private
  Stack& operator=(const Stack& s) {} //

public:
  Stack() : head(0) {}
  ~Stack();
  T pop();
  void push(T val);
  void append(T val);
};
```



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


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

```
1 template <class T,int i=128> struct Buffer{
2   T buf[i];
3 };
4
5 int main() {
6   Buffer<int> B; //i=128
7   Buffer<int,256> C;
8 }
```


Specialization

- The `class T` template parameter will accept any type `T`
- We can define a specialization for a particular type as well:

```
1  #include <iostream>
2  class A {};
3
4  template<class T> struct B {
5    void print() { std::cout << "General" << std::endl;}
6  };
7  template<> struct B<A> {
8    void print() { std::cout << "Special" << std::endl;}
9  };
10
11 int  main() {
12   B<A> b1;
13   B<int> b2;
14   b1.print(); //Special
15   b2.print(); //General
16 }
```


Templated functions

- A function definition can also be specified as a template; for example:

```
1 template<class T> void sort(T a[],
2                              const unsigned int& len);
```

- The type of the template is inferred from the argument types:
  `int a[] = {2,1,3}; sort(a,3);` $\Longrightarrow$ `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

```
1  #include <iostream>
2
3  template<unsigned int N> inline long long int fact() {
4    return N*fact<N-1>();
5  }
6
7  template<> inline long long int fact<0>() {
8    return 1;
9  }
10
11 int main() {
12   std::cout << fact<20>() << std::endl;
13 }
```


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?

