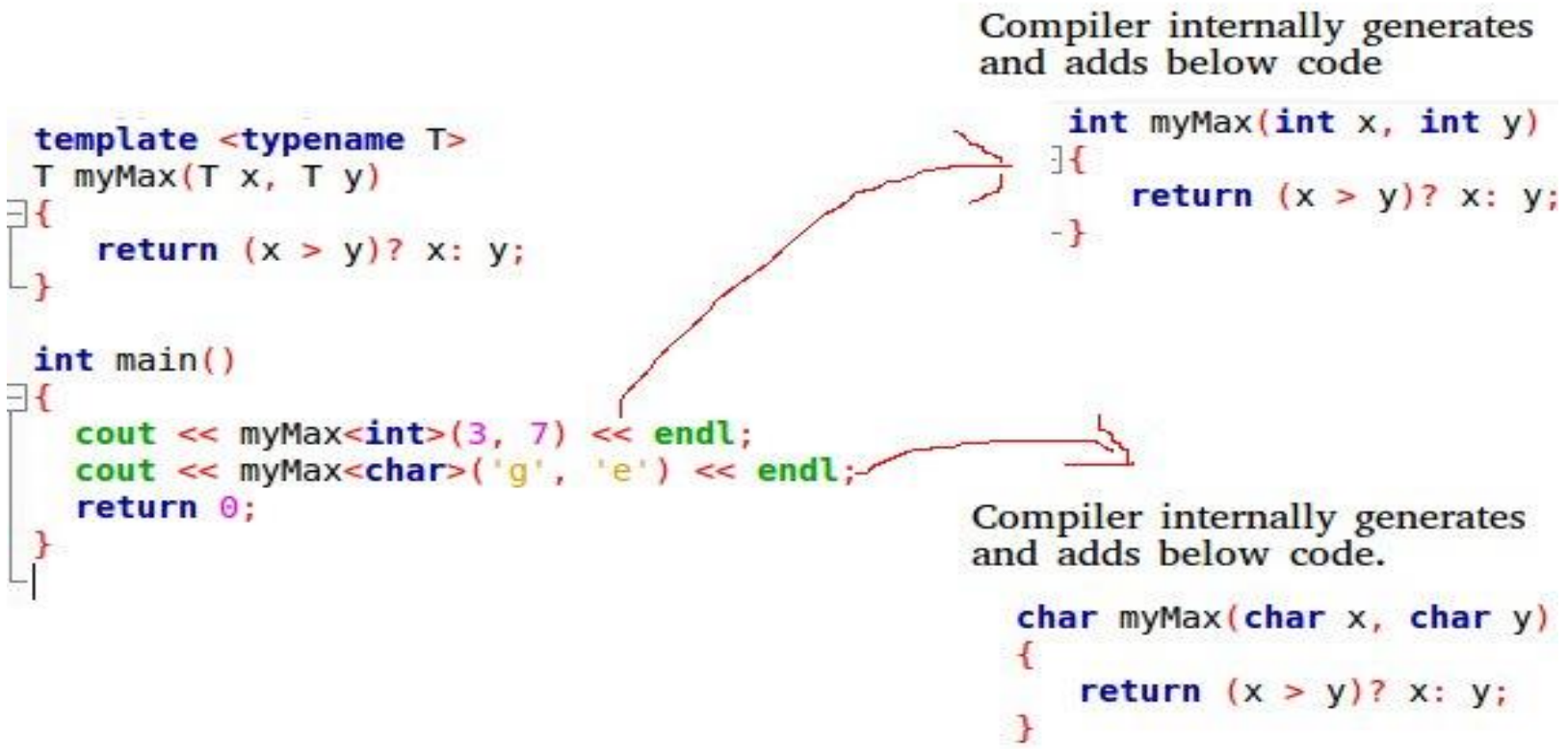
Templates, STL and RTTI Namespaces

Templates


template <typename T>
T myMax(T x, T y)
{
return (x > y)? x: y;
}
int main()
{
cout << myMax<int>(3, 7) << endl;
cout << myMax<char>('g', 'e') << endl;
return 0;
}
Compiler internally generates and adds below code
int myMax(int x, int y)
{
return (x > y)? x: y;
}
Compiler internally generates and adds below code.
char myMax(char x, char y)
{
return (x > y)? x: y;
}

Templates

- With function overloading same code needs to be repeated for different data types which leads towards waste of time & space.
- Templates enable us to define generic functions or classes which avoids above repetition of code for different data types .
- Generally templates are used if same algorithm works well for various data types eg sorting algorithms.
- There can be function templates or class templates.
- Function templates can be overloaded.

template < class type > ret type FnName (parameter list) template is a keyword used to create generic functions. class type is a placeholder

// swap function using function template

```
template <class T>
void swap (T & a, T & b)
{
T temp; temp       = a; a = b;
b = temp;
}
```

```
int main() {
    int i = 10, j = 20;
    swap( i, j );
   char ch1 = 'a', ch2 = 'b'
   swap( ch1, ch2 );
   }
```

You can define more than one generic data type in the **template** statement by using a comma-separated list.

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

template <class type1, class type2>
void myfunc(type1 x, type2 y) {
        cout << x << ' ' << y << '\n';
}

int main() {

myfunc(10, "I like C++");
myfunc(98.6, 19L);

return 0;
}
```

More generic function

```
template <class X>
void swapargs(X &a, X &b) { X temp;
temp = a; a = b;
b = temp;
cout << "Inside template swapargs.\n";
}
```

```
// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b) {
int temp; temp = a; a = b;
b = temp;
cout << "Inside swapargs int specialization.\n";
}
```

```
void main() {
swapargs(i, j); // calls explicitly overloaded swapargs()
swapargs(x, y); // calls generic swapargs(a, b);
}
```

STL (Standard Template Library)

- Provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks.
- Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.
- At the core of the standard template library are four items: *containers, algorithms, functors* and *iterators*.

Containers

Containers are objects that hold other objects.

Sequence Containers

E.g. vector, deque

Associative Containers

E.g. map

Each container class defines a set of functions that may be applied to the container.

A list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers.

They provide the means by which you will manipulate the contents of containers.

Their capabilities include initialization, sorting, searching, and transforming the contents of containers.

Common algorithms - binary_search, count, find, merge, max, min, sort

Iterators

Iterators are objects that are, similar to pointers.

Iterators give you the ability to cycle through the contents of a container.

Types of Iterators:

- Random Access
- Bidirectional
- Forward
- Input
- Output

You can increment and decrement Iterators.
You can apply the * operator to them. Iterators are declared using the **iterator** type defined by the various containers.

Functors

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors.

Functors are objects that can be treated as though they are a function or function pointer.

Vectors

The **vector** class supports a dynamic array.

you can use the standard array subscript notation to access its elements.

Any object that will be stored in a **vector** must define a default constructor.

Example:

```
vector<int> iv; // create zero-length int vector
vector<char> cv(5); // create 5-element char vector
vector<char> cv(5, 'x'); // initialize a 5-element char vector
vector<int> iv2(iv); // create int vector from an int vector
```

Vector Functions

size() – returns the current size of the vector

begin() – returns an iterator pointing to the beginning of vector

end() – returns an iterator pointing to the end of the vector

push_back() – inserts an element at the end of the vector

Insert() – used to insert an element in middle

erase() – used to remove an element form a vector

Clear() – removes all elements from the vector

Other similar containers are – List, Map, etc.

RTTI

RTTI stands for Run Time Type Identification

RTTI enables us to identify type of an object during execution of program

RTTI operators are runtime events for polymorphic classes and compile time events for all other types.

Two types of RTTI operators are there

- typeid operator
- dynamic_cast operator

typeid

- typeid is an operator which returns reference to object of type_info class
- type_info class describes type of object

typeid

```
#include<iostream>
#include<typeinfo>
using namespace std;
class A{     };
class B{     };
main() {
  int i, j;
  float f;
  char* p;
  A a;       // where A and B are classes B b;
  B b;
  cout <<"\nThe type of i is: "<< typeid(i).name();
  cout <<"\nThe type of f is: "<< typeid(f).name();  //float
  cout <<"\nThe type of p is: "<< typeid(p).name(); //char*
  cout      <<"\nThe type of a is: "<< typeid(a).name();    //class A
  cout <<"\nThe type of b is: "<< typeid(b).name(); //class B
}
```

```
main() {

employee* person; person = &sp1;
cout << typeid(person).name(); //class SalesPerson
person = &we1;
cout << typeid(person).name(); //class WageEmployee

if (typeid(we1) == typeid(we2))
// Do Something

}
```

Casting Operators

Explicit conversion is referred as cast

- const cast
- static cast
- dynamic cast
- reinterpret cast

cast operators are sometimes necessary

Explicit cast, allows programmer to momentarily suspend type checking

Syntax

```
cast_name<type>(expression);
```

casting Operators : const_cast

casts away the constness of its expression

You can not use the const_cast operator to directly override a constant variable's constant status.

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

void print (char * str) {
    cout << str << endl;
}
```

```
int main () {
const char * c = "sample text";
print ( const_cast<char *> (c) );
return 0;
}
```

casting Operators : static_cast

Any conversions which compiler performs implicitly can be made explicit using static_cast

Warning messages for loss of precision will be turned off.

reader, programmer and compiler all are made aware of fact of loss of precision.

e.g. static_cast double dval; int ival;

Casting operator: static_cast

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

class base{
  public:
     void fun() { cout<<"\n fun of base is called...";}
};

class derived:public base {
public:
    void fun2() { cout<<"\n fun2 of derived is
called..."; }
};
```

```
int main()
{
    base *ptr = new derived();
    ptr->fun();
//    ptr->fun2();    error

   //  derived *dptr = ptr;   //error invalid conversion
                                //from base * to derived *
   derived *dptr = static_cast<derived *>(ptr);
   dptr->fun2();
   dptr->fun();
    return 0;
}
/* Output:
 fun of base is called...
 fun2 of derived is called..
 fun of base is called...*/
```

casting Operators : reinterpret_cast

Complex <double> *pcom;

char *pc = reinterpret_cast < char*>(pcom)

Reinterpret cast performs low level interpretation of bit pattern

Is used to convert any data type to any other data type

The operator provides a conversion between pointers to other pointer types and numbers to pointers and vice versa.

Most dangerous

casting Operators : dynamic_cast

dynamic_cast can be used only with pointers and references to objects.

Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

dynamic_cast operators are used to obtain pointer to the derived class. dynamic_cast is always successful when we cast a class to one of its base classes.

```
#include <iostream>
#include <typeinfo> // Header for typeid operator
using namespace std;

// Base class
class MyBase {
  public:
    virtual void Print() {
        cout << "Base class" << endl;
    };
};

// Derived class
class MyDerived : public MyBase {
  public:
    void Print() {
        cout << "Derived class" << endl;
    };
};
```

```
int main()
{    // Using typeid on built-in types types for RTTI
    cout << typeid(100).name() << endl;
    cout << typeid(100.1).name() << endl;
    // Using typeid on custom types for RTTI
    MyBase* b1 = new MyBase();
    MyBase* d1 = new MyDerived();
    MyBase* ptr1;
    ptr1 = d1;
    cout << typeid(*b1).name() << endl;
    cout << typeid(*d1).name() << endl;
    cout << typeid(*ptr1).name() << endl;
    if ( typeid(*ptr1) == typeid(MyDerived) ) {
    cout << "Ptr has MyDerived object" << endl;
    }
    // Using dynamic_cast for RTTI
    MyDerived* ptr2 = dynamic_cast<MyDerived*> ( d1 );
    if ( ptr2 ) {
    cout << "Ptr has MyDerived object" << endl;
    }   return 0;  }
```

```
/*
Output -
i
d
6MyBase
9MyDerived
9MyDerived
Ptr has MyDerived object
Ptr has MyDerived object
*/
```