CPP301-C

Module 3

Using C++ Standard Library

1 Generic Programming

1.1 Generic Functions

You can have functions with the same name as long as the number of arguments are different or the type of arguments are different. This reduces the number of function names that developers have to remember. Of course you should only overload those that have similar functionality but has to provide support for multiple types. Following is a function that swaps the contents of two variables. We overloaded the function to support different types; **int**, **double** and **char *****. C++ compiler can figure out which function you are calling based on the type of arguments that you pass in.

Variable swapping functions: Generic1\main.cpp

```
#include <iostream>

// swap function for int
void swap_vars(int& v1, int& v2) {
    int vt = v1; v1 = v2; v2 = vt;
}

// swap function for double
void swap_vars(double& v1, double& v2) {
    double vt = v1; v1 = v2; v2 = vt;
}

// swap function for string
void swap_vars(const char *& v1, const char *& v2) {
    const char * vt = v1; v1 = v2; v2 = vt;
}

void main() {
    int n1 = 66, n2 = 99;
    double d1 = 1.99, d2 = 99.1;
    const char *s1 = "Hello!", *s2 = "Goodbye!";
    swap_vars(n1, n2);  // swap_vars(int&,int&)
    swap_vars(d1, d2);  // swap_vars(double&,double&)
    swap_vars(s1, s2);  // swap_vars(const char*&,const char*&)
    std::cout
        << n1 << ',' << n2 << std::endl
        << d1 << ',' << d2 << std::endl
        << s1 << ',' << s2 << std::endl;
}
```

If the code is the same but only the type is different, then you use generic functions that can be overloaded automatically for you by the compiler. In C++ you can create templates to overload generic functions. For example you may have wrote a generic function that returns the maximum between two **int** arguments.

Example int function

```
int max_value(int a, in b) { return a > b ? a : b; }
```

The above function only works with **int** type arguments. You would have to overload the above function to support another data-type. However, it will be impossible for us to overload the function to support every data-type unless we implemented a generic function instead. A generic function is defined once, and will automatically be used to generate code to support every data-types. You can use C++ templates to generate a generic function. Following example uses template to define a **max** function returning the maximum of two arguments independent of the data type. Generic functions have to be distributed in source code form usually in header files.

Defining a generic function: my_templates.h

```
#pragma once
template <typename X>
X max_value(X a,X b) { return a > b ? a : b; }
```

You can define generic arguments in the template code as data-types by using the **class** or **typename** attribute. In the above example **X** is used to indicate a data type, and we can use it anywhere in the code. At this point of time, there is no indication on what **X** is but some kind of data-type. Thus no code is generated until the template is instantiated. For a generic function, we can instantiate the template by just using the generic function directly in our code. If C++ can detect the type from arguments then there is no need to specify the exact type when calling the function.

Instantiating generic functions

```
char v1 = max_value('A','B');    // max_value<char>('A', 'B');
int v2 = max_value(10, 20);      // max_value<int>(10, 20);
double v3 = max_value(40.20, 18.00);// max_value<double,double>(40.20,18.00);
```

What about operations that would require using more than one type? You can indicate as many generic type arguments as you like in a template. Thus you may have more than one type but they can also be the same type. The types can be used anywhere in the function.

Multiple generic types

```
template <typename X,typename Y>
X add_any(X arg1,Y arg2) { return (X)(arg1 + (X)arg2); }
```

Instantiating with multiple types

```
char a = add_any('A', 2);        // add_any<char,int>('A', 2);
long b = add_any(10, 69.99);     // add_any<int,double>(10, 69.99);
```

Following shows the generic version of the swap variable function. There is only one generic argument since the function only needs one type as both arguments and local variable is the same time. You can now able to swap any kind of variables by just including the above header file and call the function.

Defining a generic swap function: my_templates.h

```
template<typename T>
void swap_vars(T& v1, T& v2) {
    T vt = v1; v1 = v2; v2 = vt;
}
```

Using generic function

```
swap_vars<int>(n1, n2);      // swap_vars(int&,int&)
swap_vars<double>(d1, d2);   // swap_vars(double&,double&)
swap_vars<(s1, s2);          // swap_vars(const char*&,const char*&)
```

Generic arguments do not have to be types. Following shows how we can capture the size of an array which is an **int** value. Note that arrays are passed by pointers and not by value or reference. To get the size of an array and be able to change the array you need to change it to pass by reference.

Non-type generic arguments

```
template<typename T, int SIZE>
void display_array(T (&a)[SIZE]) {
    for (int i = 0; i < SIZE; i++)
        std::cout << a[i] << std::endl;
}
```

Inferring the type and size of arrays from arguments

```
void main() {
    int a1[] = { 10, 20, 30, 40, 50 };
    double a2[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    char a3[] = { 'A', 'B', 'C', 'D', 'E', 'F' };
    display_array(a1);
    display_array(a2);
    display_array(a3);
}
```

1.2 Generic Classes

Templates can be used to implement both generic functions and generic classes as well. Most applications would probably need to use standard data structures such as binary trees, linked-lists, stacks, queues and so on. It would be tedious for each programmer to have to build all these from scratch even though these kinds of data structures have already been implemented countless of times before. The example below uses a template to create a simple stack classes for different data types.

Defining a generic class: Generic2\Stack.h

```
template <typename T>
class CStack {
protected:
    T m_values[256];
    int m_count;
public:
    CStack() : m_count(0) { }
    bool Empty() { return m_count == 0; }
    int GetCount() const { return m_count; }
    void Push(T value) {
           if(m_count == 256) throw "Stack is full.";
           m_values[m_count++] = value;
    }
    T Pop() {
        if(m_count == 0) throw "Stack is empty.";
        return m_values[--m_count];
      }
};
```

No code is generated if templates are not instantiated. To instantiate a class template, use the class within the code. When the template is used, the actual data type must be specified.

Instantiating class templates: main.cpp

```
CStack<char> stack1;
CStack<int> stack2;
CStack<double> stack3;
CStack<char *> stack4;
stack1.Push('A'); stack1.Push('B'); stack1.Push('C');
stack2.Push(100); stack2.Push(200); stack2.Push(300);
stack3.Push(1.1); stack3.Push(2.2); stack3.Push(3.3);
stack4.Push("Hello!"); stack4.Push("Goodbye!"); stack4.Push("Sayonara!");
while(!stack1.Empty()) std::cout << stack1.Pop() << std::endl;
while(!stack2.Empty()) std::cout << stack2.Pop() << std::endl;
while(!stack3.Empty()) std::cout << stack3.Pop() << std::endl;
while(!stack4.Empty()) std::cout << stack4.Pop() << std::endl;
```

You can also declare non-type arguments for a template. This can be used to replace values in the code and not just to replace a data-type. In the following example, we modified the template declaration to allow the class user to determine the maximum size of the stack. You can also assign default values to arguments.

Defining non-type parameters

```
template <typename T = int,int SIZE = 256>
class CStack {
protected:
    T m_values[SIZE];
        :
    void Push(T value) {
           if(m_count == SIZE) throw "Stack is full.";
           m_values[m_count++] = value;
     }
};
```

Instantiating templates with non-type or default parameters

```
CStack<char,128> stack1;    // not using defaults
CStack<> stack2;            // using all defaults
CStack<double> stack3;      // using default SIZE
```

You can use **typedef** to pre-instantiate specific classes from a template. You will not realize that some types are actually generated from templates since the angle braces will not be used for pre-instantiated types.

Using typedefs to pre-instantiate generic types

```
typedef CStack<double> DoubleStack;
typedef CStack<int, 52> CardStack;
```

Using pre-instantiated generic types

```
DoubleStack stack5;
CardStack s6;
```

1.3 Dynamic Memory

Note that a different class is generated if the generic arguments is different. Having to generate a new class just because the array size is different is not advisable. As they are different classes they are not compatible even though they store the same type of data. Any code written for one class would not work for another incompatible class. It is better to allocate a dynamic array rather than a fixed size array. However managing dynamic memory has to be done properly to avoid memory leaks. You can start off by creating a template for a generic stack class that uses a **m_values** pointer to store the address of an allocated block of dynamic array. A **m_capacity** member stores the size of the dynamic array in number of elements rather than bytes. You can easily get the size in bytes by multiplying m_capacity with the **sizeof** the generic type.

Generic class with member variables: Generic2\my_templates.h

```
template <typename T>
class CStack {
protected:
    T *m_values;
    int m_capacity;
    int m_count;
}
```

In the constructor you can allocate a default capacity of 256 elements for the array. Use **new** to allocate dynamic memory for the array that returns its memory address to be stored for access and deletion later.

Constructor and destructor to allocate and release dynamic memory

```
public:
    CStack() : m_capacity(256), m_count(0) { m_values = new T[m_capacity]; }
    ~CStack() { delete m_values; }
```

To support passing the object by value, you definitely need to provide your own copy constructor as the default copy constructor would only copy the pointer and will not physically allocate a new block of memory and copy the data over.

Copy constructor to physically make a copy of dynamic memory

```
CStack(CStack& obj) : m_capacity(obj.m_capacity), m_count(obj.m_count) {
    size_t size = m_capacity * sizeof(T);
    m_values = new T[m_capacity];
    memcpy_s(m_values, size, obj.m_values, size);
}
```

To be able to store any number of elements up to available memory and the limits set by the data type and operating system, you can resize the dynamic array to fit more items if the current capacity has been exceeded or also to fit less items. A new array is allocated and existing items will be copied over from the old array. The old array is then deleted and the capacity updated.

Resizing by allocating a new array and copying from the old array

```
void Resize(int capacity) {
    size_t old_size = m_capacity * sizeof(T);
    size_t new_size = capacity * sizeof(T);
    if (old_size > new_size) old_size = new_size;
    if (m_count > capacity) m_count = capacity;
    T *values = new T[capacity];
    memcpy_s(values, old_size, m_values, old_size);
    delete m_values; m_values = values;
    m_capacity = capacity;
}
```

Rest of the functions

```
bool Empty() { return m_count == 0; }
int GetCount() const { return m_count; }
void Push(T value) {
//  automatically doubles the capacity up to limit of 1 million
    if (m_count == m_capacity) {
        int capacity = m_capacity * 2;
        if (capacity > 1000000) capacity = 1000000;
        if (m_capacity == capacity) throw "Stack is full"; // already 1 million
        else Resize(capacity);
    }   m_values[m_count++] = value;
}
T Pop() {
    if (m_count == 0) throw "Stack is empty.";
    return m_values[--m_count];
}
```

Testing dynamic stack

```
#include <iostream>
void main() {
    CStack<int> s1;
    for (int index = 0; index < 1000; index++) s1.Push(index);
    while (!s1.Empty()) std::cout << s1.Pop() << std::endl;
}
```

1.4 Implementing Nullable

In C/C++, only pointers can be null. Sometimes a function may use null as indicator to use a default value. For this example we can implement a generic **nullable** type where it may or may not be assigned a value at construction.

Implementing a generic nullable type: MyTypes.h

```
template<typename T>
class nullable {
private:
    T m_value;
    bool m_hasValue;
public:
    nullable() { m_hasValue = false; }
    nullable(void *ptr) {
        if (ptr) throw std::exception("not null"); m_hasValue = false; }
    nullable(T value) { m_value = value; m_hasValue = true; }
    bool hasValue() const { return m_hasValue; }
    operator T() const {
        if (!m_hasValue) throw std::exception("null"); return m_value; }
};
```

Function that uses a nullable int

```
void show(my::nullable<int> v = 123) {
    if (v.hasValue()) cout << v << endl;
}


void main() {
    show();         // use default value
    show(nullptr);  // explicit null
    show(999);      // explicit value
}
```

2 Standard Template Library

2.1 STL Elements

STL core elements are **containers**, **iterators** and **algorithms**. Containers determine how data is stored and in what manner the data can be retrieved. Iterators allow you to access the data stored in the container using the concept of pointers. Every time you increment iterators, you can retrieve the next item while decrementing it will allow you to access the previous item. Algorithms are pre-defined type of operations that can be used on any containers, such as sorting. The STL templates are declared in the **std** namespace. Thus you have to prefix templates used from the STL with this namespace. However, you can use the **using namespace** to import the elements from this namespace.

Importing STL templates into the global namespace

```
using namespace std;
```

There are many different types of containers provided in STL. Following shows the **vector** container. The problem with using arrays is that once you allocate the space for an array, the size is fixed. You can only store and access elements in array but you cannot add more elements or remove existing elements from an array. A vector is a STL container to replace the array. It is dynamic as you can choose to add or remove items at any time in the container. Vectors are used to store a list of items. You can treat a vector as a stack, since it allows both push and pop operations. Another nice feature is that you can access items in a vector as an array. To add or remove items in a vector, you can use **push_back** and **pop_back** functions. You can then use a **size** function to find out how many elements are within the vector.

Adding and removing elements from a vector: STL1\main.cpp

```
#include <iostream>
#include <vector>

using namespace std;

void main() {
    vector<int> v;
    v.push_back(100);       // append element
    v.push_back(200);
    v.push_back(300);
    cout << v.size() << endl;
    v.pop_back();           // remove element
    cout << v.size() << endl;
}
```

Since a vector can be treated as an array, you can retrieve elements from the vector using the array (**[]**) operator. Use **front** and **back** functions to specifically retrieve the first and last elements in the vector. Since operators and functions return references rather than values, you can use it to modify the elements within the vector. You can use **clear** to erase all elements and **empty** to check whether there are any elements in the container.

Retrieving elements using array operator and functions

```
for(int i=0;i<n.size();i++) cout << v[i] << endl;
cout << v.front() << endl;
cout << v.back() << endl;
```

Modifying elements retrieved

```
v.front() = 99;
v.back() = 66;
```

Clearing the vector

```
v.clear();
if(v.empty()) cout <<
      "No elements!" << endl;
```

A vector by default only allows you to append or remove items from the back. If you need to be able to add and remove items from both back and front of the container you can use a deque (double-ended queue) instead. A double-ended queue container allows access from both sides as it has functions such as **push_back**, **push_front**, **pop_back**, **pop_front** to allow you to add and remove elements from both front and back. Use **front** and **back** functions to retrieve the value just like a vector.

Using deques

```
#include <iostream>
#include <vector>
#include <deque>

using namespace std;

void main() {
        :
    deque<char *> d;
    d.push_front("Hello!");
    d.push_back("Goodbye!");
    cout << d.front() << endl;
    cout << d.back() << endl;
    d.pop_back();
    d.pop_front();
        :
}
```

Iterators allow us to access elements within a container the same way we can use pointers to access elements. You need to declare the iterators through the container template to generate the correct iterator type or just use **auto**. Use **begin** method to retrieve an iterator that points to the first element, or **end** to retrieve iterator for end of the container. You can then use increment and decrement operators to move the pointer through the elements.

Using iterators

```
vector<int>::iterator b1 = v.begin();    // auto b1 = v.begin();
vector<int>::iterator e1 = v.end();      // auto e1 = v.end();
while(b1 != e1) cout << *(b1++) << endl;
```

While **begin** and **end** retrieves a pointer to first and past the last element, you can also obtain a reverse iterator, where data are always accessed in reverse order, even though the same operators are used. Use the **rbegin** and **rend** functions to retrieve reverse iterators.

Using reverse iterators

```
vector<int>::reverse_iterator b2 = v.rbegin();   // auto b2 = v.rbegin();
vector<int>::reverse_iterator e2 = v.rend();     // auto e2 = v.rend();
while(b2 != e2) cout << *(b2++) << endl;
```

To iterate through containers is now much simpler in C++/11. Beforehand we need to explicitly call **begin** function to retrieve the first iterator and **end** function to get the end of the container. The iterator also has to be dereferenced to access the value. The interator is increment to point to the next object until the end is reached. All this code can now be replaced by the following new iterator **for** loop in C++/11.

Iterating through a container in C++/11

```
for(auto n : v) cout << n << endl;
```

Even though certain containers are not efficient when performing certain types of operations, these operations may still be available as long as containers have support for iterators. A vector allows you to insert and delete elements at any position even though it is not efficient for this type of container. Use **insert** and **erase** function to perform such operations by using iterators.

Inserting elements

```
v.insert(v.begin(),160);
v.insert(v.begin() + 1, 480);
v.insert(v.end() - 1,200);
```

Algorithms are generic functions that can be applied to containers to perform specific processing. Algorithms commonly use iterators since most containers have support for them. Following is some example algorithms. A **sort** function is also provided to sort elements in containers that do not have their own sorting support. You need to provide iterators specifying the range of elements to sort. In the following example, we sort a list of numbers within a vector. If the elements have been sorted, you can use the **binary_search** algorithm function to perform faster searching operation on containers. The following is an example of binary searching. Elements can be reversed or rotated using **reverse** and the **rotate** functions.

Sorting a vector

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

void main() {
    vector<int> v(4);
    v[0] = 99;
    v[1] = 62;
    v[2] = 110;
    v[3] = 103;
    sort(v.begin(),v.end());
    for(auto n : v) cout << n << endl;
}
```

Other algorithms

```
sort(v.begin(),v.end());
if(binary_search(v.begin(),v.end(),110))
    cout << "Found!" << endl;
reverse(v.begin(),v.end());
rotate(v.begin(),v.begin()+2,v.end());
```

2.2 String

In C/C++, strings are actually character arrays that contain a NULL character to mark the end of the string. In modern programming languages, strings are implemented as objects. In STL there is a **string** class to create objects to store and process ANSI strings. For Unicode strings, use **wstring** class instead. However you will also need to use **wcout** and **wcin** for console input and output of unicode strings even though the console window can only support ANSI strings.

Using strings: String1\main.cpp

```
#include <iostream>
#include <string>
using namespace std;
void main() {
      string s1 = "Hello!"; s1.append(" Goodbye!"); cout << s1.size() << endl;
      s1.replace(7, 7, "Ciao"); s1.erase(0, 7); s1.insert(0, "Sayonara! ");
      cout << s1.c_str() << endl;
      wstring s2 = L"Hello!";
      wcout << s2.c_str() << endl;
}
```

2.3 Maps & Sets

These containers are associative containers that allows you to use a key to associate with elements. You can then fetch the element by using the key. The main difference between **map** and **multimap** is that **map** requires unique keys while **multimap** allow duplicate keys. A pair template is required to construct the element that will consist of the key and value to add into map containers. Use **insert** function to add paired elements into a map. You can then use the **find** method to locate the element using the key that returns an iterator to the paired element. From a paired element, use the **first** attribute to access the key, and **second** to access the value. In a **multimap**, you can continue to increment the iterator to retrieve the next matching item until you reach one that does not match by checking the iterator against the one retrieved by **upper_bound**.

A **set** is like a **map** except the value of the element will also be used as the key to locate the element. This can be used to determine whether a value is part of a **set** or not. Elements in a **set** must be unique while it can be repeated in a **multiset**.

Using maps: Maps1\main.cpp

```
#include <iostream>
#include <map>
using namespace std;
void main() {
    map<int,string> m;
    m.insert(pair<int,string>(100,"Can of coke"));
    m.insert(pair<int,string>(200,"Bottle of ketchup"));
    m.insert(pair<int,string>(300,"Pack of noodles"));
    map<int,string>::iterator p = m.find(200);
    if(p == m.end()) {
        cout << "Element not found!" << endl; return; }
    cout << "Key: " << p->first << endl;
    cout << "Value : " << p->second << endl;
}
```

Using sets: Sets1\main.cpp

```
#include <iostream>
#include <set>
using namespace std;
void main() {
    set<string> s;
    s.insert("Rice Cooker");
    s.insert("Food Blender");
    s.insert("Television");
    set<string>::iterator p;
    p = s.find("Television");
    if(p == s.end()) cout << "Not found!" << endl;
}
```

2.4 STL Objects

The STL templates can be instantiated to store any type including objects. If you are using a container that can sort or search for data such as a priority queue, maps and sets, you need to overload the less than operator that will be called by container for such operations. Following example demonstrates storing and searching for objects in a set.

Implementing STL compatible objects: STLObjects1\main.cpp

```
#include <set>
#include <string>
#include <iostream>

using namespace std;

class CContact {
    string m_name;
    string m_email;
public:
    CContact(const char *name, const char *email)
        : m_name(name), m_email(email) { }
    CContact(const string& name, const string& email)
        : m_name(name), m_email(email) { }
    string GetName() const { return m_name; }
    string GetEmail() const { return m_email; }
    friend bool operator < (const CContact& c1, const CContact& c2) {
        return c1.m_name < c2.m_name; }
};
```

The above is the format required for overloading less than operator. You must follow the format as shown to work with STL. Overload globally but mark it friend function if you need to access the private members from the function. Note that in our example, we only make use of the contact name for sorting and searching.

Storing and searching objects

```
multiset<CContact> contacts;
contacts.insert(CContact("Phillip","the_visualizer@yahoo.com"));
contacts.insert(CContact("Phillip","visualizer@domaindlx.com"));
contacts.insert(CContact("Sally","sasa88@hotmail.com"));
char name[64]; cout << "Enter name:";
cin.getline(name,sizeof(name));
CContact contact(name,"");
multiset<CContact>::iterator p = contacts.find(contact);
if(p != contacts.end()) {
    multiset<CContact>::iterator e = contacts.upper_bound(contact);
    do { cout << p->GetName() << ',' << p->GetEmail() << endl;
    } while (++p != e);
}
```

2.5 Extending & Aggregating Containers

There are two ways to re-use existing types; inheritance and aggregation. When you extend a new class from a base class, you gain all the features of the base class. You can then use those features in your new class through inheritance. Another choice is for the new class to contain an object from an existing class. When the outer object is created, the inner object will also be created. The outer constructor will be extending the inner object constructors as well. When the outer object is destroyed, the inner object will also be destroyed as well. The outer destructor will be extending the inner object destructor. This is called aggregation. All you need to do is to write code in the outer object to use or expose the inner object.

Reuse through inheritance: Aggregate1\main.cpp

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

class List1 : vector<double> {  // List1 is a vector<double>
public:
    List1() {   // List1 : vector<double>()
        push_back(1.1);
        push_back(2.2);
        push_back(3.3);
    }
    double Sum() {
        double total = 0;
        for (auto value : *this)    // requires C++/11
            total += value;
        return total;
    }
};
```

Reuse through aggregation

```
class List2 {   // List2 aggregates a vector<double>
private:
    vector<double> m_v;
public:
    List2() {   // List2 : m_v()
        m_v.push_back(1.1);
        m_v.push_back(2.2);
        m_v.push_back(3.3);
    }
    double Sum() {
        double total = 0;
        for (auto value : m_v)  // requires C++/11
            total += value;
        return total;
    }
    void push_back(double v) {
        m_v.push_back(v);
    }
};

void main() {
    List1 l1;
    List2 l2;
    l1.push_back(4.4);
    l2.push_back(4.4);
    cout << l1.Sum() << endl;
    cout << l2.Sum() << endl;
}
```

2.6 Exceptions

Even though there is no standard built-in exception class in C++, the standard library does provide an exception class. All the C++ containers iterators and templates throw objects from this class. You can access the error message through the **what** member function that returns a c-style string. A basic exception can only store a message but you can extend the class to store additional information.

Throwing exception by value

```
try { throw exception("throwing C++ exception."); }
catch (exception& e) { cout << "Exception caught : " << e.what() << endl; }
```

Throwing dynamic exception

```
try { throw new exception("throwing C++ exception."); }
catch (exception* e) {
    cout << "Exception caught : " << e->what() << endl;
    delete e;   // destroy dynamic object
}
```

Rather than implemently completely new types you can always extend existing types. For example rather than implementing your exception class, you can extend the C++ standard library exception class and add missing features. It also guarantees that it is compatible to the base class.

Extending exceptions

```
#include <exception>

class my_exception : public std::exception {
    std::string m_file;
    int m_line;
public:
    my_exception(
        const char * message,
        const char * file, int line)
        : std::exception(message), m_file(file), m_line(line) { }
    const std::string& file() const { return m_file;  }
    int line() const { return m_line; }
};
```

Throwing a custom exception

```
throw my_exception("Error message", __FILE__, __LINE__)
```

3 Object Dependency

3.1 Runtime-Type Information

Dynamic cast uses Runtime-Type Information generated by the compiler for the types that are polymorphism. At runtime the virtual-table assigned to each object is used to retrieve type-information about the actual object. Use **typeid** operator to access the **type_info** from any type, variable or object. Use the **name** function to return a text description of the type. Comparison operators are overloaded to allow you to compare type information to see whether they are the same type or different type.

A root class: TypeInfo1

```
class A {
private:
    int m_value1;
public:
    A() : m_value1{} {}
    virtual void func1(int v) { m_value1 = v; }
};
```

A derived class from base class A

```
class B : public A {
private:
    int m_value2;
public:
    B() : m_value2{} {}
    void funcA(int v) { func1(v * 2); }
    virtual void func2(int v) { m_value2 = v; }
};
```

A derived class from base class B

```
class C : public B {
private:
    int m_value3;
public:
    C() : m_value3{} {}
    virtual void func3(int v) { m_value3 = v; }
};
```

A separate root class

```
class D {
private:
    int m_value1;
    int m_value2;
    int m_value3;
public:
    D() { m_value1 = m_value2 = m_value3 = 0; }
    virtual void func4(int v1, int v2, int v3) {
        m_value1 = v1;
        m_value2 = v2;
        m_value3 = v3;
    }
};
```

Accessing type information

```
#include <typeinfo>

void main() {
    A *p1 = new B();
    A *p2 = new C();
    cout << typeid(p1).name() << endl;  // p1 is a A*
    cout << typeid(p2).name() << endl;  // p2 is a A*
    cout << typeid(*p1).name() << endl; // *p1 is a B object
    cout << typeid(*p2).name() << endl; // *p2 is a C object
}
```

Comparing type_info references

```
if (typeid(p1) == typeid(p2)) cout << "pointers are the same." << endl;
if (typeid(*p1) != typeid(*p2)) cout << "objects are different." << endl;
```

Comparing against specific types

```
if (typeid(*p1) == typeid(B)) cout << "p1 points to a B object." << endl;
if (typeid(*p2) != typeid(B))
    cout << "p2 does not point to a B object." << endl;
if (typeid(*p2) == typeid(C)) cout << "p2 points to a C object." << endl;
```

If you wish to do multiple comparisons on type info, you should assign the reference returned back from **type_id** to a variable. You can then use the variable to reference the type information multiple times. To check class hierarchy and compability, call the **before** function. Full compatibility is guaranteed if the result is **true**. You should only use objects that are completely compatible.

Checking compatibility at runtime

```
const type_info& i1 = typeid(*p1);
const type_info& i2 = typeid(*p2);
```

```
if (i1.before(i2)) cout << "B comes before C." << endl;
if (!i2.before(i1)) cout << "C does not come before B. " << endl;

const type_info& i3 = typeid(A);
if (i3.before(i1)) cout << "B is fully compatible to A." << endl;
if (i3.before(i2)) cout << "C is fully compatible to A." << endl;
```

3.2 Use C++ Casting

Explicit casting is not required if the compiler knows the exact type of object and can determine whether they are compatible or not. However when using objects that are not fully compatible or the compiler has no idea what if the exact type of object, you need to cast a reference or pointer to the object to use the object. Casting is available since C, however it is preferable to use C++ casting operators instead. The following is a set of classes where some are compatible and not compatible.

Class compatibility with polymorphic pointers

```
A *p1 = new A();
A *p2 = new B();     // B compatible with A
A *p3 = new C();     // C compatible with A
B *p4 = new C();     // C compatible with B
D *p5 = new D();
B *p6 = p1;          // A is only partly-compatible with B
```

Assignment from **A*** to **B*** is not allowed since full compatibility is not guaranteed. If you use only functions in **B** that access only variables in **A**, then it would still work but accessing variables that does not exist would crash the program. You can typecast to force the compiler to make the conversion between the incompatible types.

Typecasting partly-compatible pointers

```
    B *p6 = (B *)p1;     // partly-compatible
    p6->funcA(66);       // works
//  p6->func2(99);       // crash
```

Typecasting incompatible pointers

```
    B *p7 = (B *)p5;     // incompatible
    p7->funcA(66);       // crash
```

However C-style casting does not perform any compatibility check before casting. This means that you can also cast types that are completely incompatible as shown below. Possible effects is performing the wrong operation, memory corruption and messing up the call stack causing potential program crash. Instead of using C-style casting, you can use C++ casting operators. There are a few available; **reinterpret_cast**, **static_cast** and **dynamic_cast**. Using reinterpret_cast does not solve over problem though since it is just a C++ version of C-style casting. No checking is done on compatibility.

The static_cast operator checks to make sure at least partial-compatibility exists. The dynamic_cast operator is specially built for use with polymorphism so its check compatibility at runtime and returns a **nullptr** if it is not compatible.

C++ version of C-style casting

```
B *p6 = reinterpret_cast<B *>(p1);
B *p7 = reinterpret_cast<B *>(p5);
```

Checking partial-compatibility at compilation time

```
B *p6 = static_cast<B *>(p1);
B *p7 = static_cast<B *>(p5);
```

Checking compatibility at runtime

```
B *p6 = dynamic_cast<B *>(p3);
B *p7 = dynamic_cast<B *>(p5);
if (p6 != nullptr) cout << "p6 is compatible to B" << endl;
if (p7 == nullptr) cout << "p7 is not compatible to B" << endl;
```

There is also a **const_cast** that should be avoided. This operator is used to cast-away the **const** to make references and pointers that can then be used to update read-only values. However accessing the constant would probably give you back the old value due to caching. You can mark variables as **volatile** to disable caching. Using this cast defeats the purpose of implementing constants.

Updating constant values

```
    const int n1 = 66;  //  n1 = 99; // cannot compile
    const_cast<int&>(n1) = 99;  // works
    cout << n1 << endl;         // still 66 because of caching
    cout << const_cast<int&>(n1) << endl;  // 99
```

Marking constant as volatile

```
    const volatile int n1 = 66;
```

3.3 Object Repository

In a large application there can be hundreds or thousands of objects at a time created from classes in different libraries. These libraries could also be implemented by many different developers. The problem is that these objects are not standalone, they need to communicate and cooperate with each other. They are like components that needs to be integrated together into a single system. They depend on each other but has to be loosely coupled so that they can be added, removed and replace at any time. What you need is to have some sort of container to resolve dependencies between objects.

The following is a simple implementation of an object repository where we can place objects that we wish to share and other functions can fetch objects they wish to use. The class can use a **map** to store **pair** objects that contains the type name and a pointer to the object of that type. We can add an **Inject** function to register an object and **Lookup** to access the object.

To be able to store and retrieve back any kind of object, we implement the functions as generic. We can also use runtime-type information to fetch the type of object and register the object in the **map** using the type name so the type name can be used to fetch back the object. You can now see from the example below how easy it will be to share objects with the object respository. We register a logger into the repository by using the **ILogger** type and the same type can then be used to get access to the logger.

Declaring an object repository class: Repository1\Logger.h

```
class CServiceRepository {
private:
    std::map<const char *, void *> m_services;
public:
    template<typename T>
    void Inject(T* service) { auto serviceType = typeid(T).name();
        m_services.insert(std::pair<const char *, void *>(
            serviceType, service));
    }
    template<typename T>
    T* Lookup() { auto serviceType = typeid(T).name();
        auto p = m_services.find(serviceType);
        if (p == m_services.end()) return nullptr;
        return reinterpret_cast<T *>(p->second);
    }
};
```

Sharing objects through repository

```
CServiceRepository repository;

void log() {
    auto logger = repository.Lookup<ILogger>();
    if (logger != nullptr) logger->Message(_T("Hello!"));
}

void main() {
    CDefaultLogger logger;
    repository.Inject<ILogger>(&logger);
    log();
}
```

The code that places the pointer to a logger into the repository and the code that uses the logger does not need to be in the same module. One part can be in the EXE or in a DLL and the other part can also be in a separate DLL. The EXE and DLL can be built by different programmers. The other thing they need to know about or agree on is to use the same interface. If you use an object through an interface you do not need to know the class nor where the class is implemented. As long as you can fetch a pointer to the interface table you can use the object. And if the object does not exist it does not mean that your program can no longer operate. For example if there is no logger, then we simply do not do logging or we can still use a default logger instead. You can thus build fault-tolerant software systems.