Akademin

Generic Programming

C++ Language

Generic Programming

- Generic programming is a programming style to make reusable, generic and type independent classes, functions, algorithms and etc.
- C++ supports generic programming using template parameters

    template <comma-separated-list-of-parameters> Declaration

- Using template we can have ***static polymorphism***, ***function***, ***class***, ***variable***, ***alias***, ***variadic*** templates and the Standard Template Library (STL) is a collection of template containers, iterators, algorithms, etc. The C++ standard library is mostly template based.

    [C++ Standard Library headers]

- In the example, the template parameter type is T.

```cpp
#include <iostream>
template <typename T>
T max(T x, T y)  {  return (x > y) ? x : y; }

int main(void) {
   std::cout << max(2, 3) << std::endl;
   std::cout << max(5.9, 6.7) << std::endl;
   return 0;
}
```

Generic Programming

- The process of replacing template parameters by concrete types is called ***instantiation***
  - In the example, two instances of the function is generated, (for int and double)
- Templates are compiled in two phases:
  - At **definition time**(without instantiation), the template code is checked regardless of the template parameters. E.g. syntax errors and etc.
  - At **instantiation time**, the template code is checked to ensure that the code is valid.
    - E.g. if x or y in the max function does not support the comparison operator, then an error is generated.
- Templates should be declared and defined in the same translation unit.
  - As the best practice declare and define a template in a header file.

```
#include "module.h"
#include <iostream>

int main(void) { print(3); print("Hello"); return 0; }
```

```
#ifndef MODULE_H
#define MODULE_H
template <typename T>void print(T value);
#endif
```

```
#include "module.h"
#include <iostream>

template <typename T>
void print(T x) { std::cout << x << std::endl; }
template void print(int);
template void print(char const *);
```

Generic Programming

- Possible to have multiple template parameters. E.g. **template <typename T, typename U> ...**
- Generally template parameters are determined by the arguments we pass
  - E.g. in **max(2, 6)**, T is deduced to int. Because **2** and **6** are of type int.
- Type conversions during type deduction are limited
  - Qualifiers (const and volatile) are ignored
  - References are converted to the referenced type
  - Raw arrays or functions are converted to the corresponding pointer type
  - Explicitly we can convert or specify the type(s)
    - max<**double**>(4, 7.2);
    - max(**static_cast<double>(4)**, 7.2);

```cpp
#include <iostream>

template <typename T>
T max(T x, T y) { return (x > y) ? x : y; }

int main(void) {
  int i = 10; int &ir = i; int arr[4]{}; const int c = 42;

  max(i, c);    // T is deduced as int
  max(c, c);    // T is deduced as int
  max(i, ir);   // T is deduced as int
  max(&i, arr); // T is deduced as int*
  max(4, 7.2);  // Error: T can be deduced as int or double

  return 0;
}
```

Generic Programming

- Possible to have default types for template parameters. E.g. `template <typename T = int>`
  - In the case of type deduction for default arguments specify the default types
- It is possible to deduce return types using auto. E.g.
  - `template <typename T, typename U> auto func(T a, U b);`
  - `template <typename T, typename U> auto func(T a, U b) -> decltype(b * a) { return b * a; }`
  - If it is required, type casting shall be used
- Possible to have explicit function template specialization
- There are a lot of standard template functions

```
template <typename T = std::string>
void print(T x = "") { std::cout << x << std::endl; }
int main(void) { print(1); print(); return 0; }
```

```
template <typename T>
void print(T x) { std::cout << x << std::endl; }
template <> void print(double x) {
    std::cout << std::scientific << x << std::endl;
}
```

```
int array[5]{3, 5, 1, 8, 0};
std::sort(array, array + 5, [](auto m, auto n) { return (m > n); });
for (const auto &elem : array) { std::cout << elem << "\t"; }
```

Generic Programming

- Possible to have non-type template parameters and even default values. E.g.
  - template <typename T, std::size_t SIZE = 10> …
  - template <int Val, typename T> or template <typename T, T Val = T{}>
  - In C++ 17, auto can be used to deduce type of a non-type parameter.
  - A non-type parameter shall be a compile-time constant
  - Type of a non-type parameter can only be an integral, an enumeration type, std::nullptr_t, a pointer or reference to a class object or a function
- Possible to make variable and alias templates

```
template <typename T>constexpr T PI = T(3.1415926535'8979323846'2643383279'5028841972L);
int main(void) { std::cout << PI<float>; return 0; }
```

- Possible to check types using type traits. For example: std::is_arithmetic<T>::value

```
template <typename T> using constptr_t = const T *; // A template const pointer
int main(void) { constptr_t<int> ptr{nullptr}; return 0; }
```

Generic Programming

- Possible to disable templates and have conditional instantiation using **[std::enable_if<>]()**
  - template <bool B, typename T = void> struct enable_if;
- Like template functions, it is possible to make template and generic classes
  - E.g. template <typename T> class Point { T x, y; };
- Declaration and definition of a template class shall be in the same file (a header file)
- Possible to have class template specialization
- Possible to have class template partial specialization

```
#include <iostream>

template <typename T>
class Point {
private: T x, y;
public:
   Point(T m = 0, T n = 0) : x{m}, y{n} {}
   T getX(void) { return x; }
   T getY(void) { return y; }
   void print(void);
};

template <typename T>
void Point<T>::print(void)
{ std::cout << "(" << x << ", " << y << ") "; }

int main(void) {
   Point<int> p{1, 2}; // An int type instance
   Point<double> q{2.5, 3.5}; // A double type instance
   p.print();
   q.print();
   return 0;
}
```

Generic Programming

```cpp
/** Possible to have function template specialization **/
#include <iostream>

template <typename T>
class Point {
private: T x, y;
public:
    Point(T m = 0, T n = 0) : x{m}, y{n} {}
    T getX(void) { return x; }
    T getY(void) { return y; }
    void print(void) { std::cout << "(" << x << ", " << y << ") "; }
};

template <>
void Point<double>::print(void) // Specialized for double
{ std::cout << std::scientific << "(" << x << ", " << y << ") "; }

int main(void) {
    Point<int> p{1, 2};        // An int type instance
    Point<double> q{2.5, 3.5}; // A double type instance
    p.print(); q.print();
    return 0;
}
```

```cpp
/* A non-type parameter can have a default value */
#include <iostream>
template <typename T, int SIZE = 8>
class Stack {
private: int top{-1};  T stack[SIZE]{};
public:
  Stack() { static_assert(SIZE > 0); }
  bool push(const T &value) {
    bool status = false;
    if (top < (SIZE - 1)) { top++; status = true; stack[top] = value;  }
    return status;
  }
  bool pop(T &elem) {
    bool status = false;
    if (top > -1) { elem = stack[top]; status = true; top--; }
    return status;
  }
};
int main(void) {
  Stack<int, 8> stack;
  int value = 10; (void)stack.push(value);
  value = 0; (void)stack.pop(value);  std::cout << value << std::endl;
  return 0;
}
```

Generic Programming

❖ Possible to have class template specialization

```cpp
#include <iostream>

template <typename T>
class Point {
private:
    T x, y;
public:
    Point(T m = 0, T n = 0) : x{m}, y{n} {}
    T getX(void) const { return x; }
    T getY(void) const { return y; }
    void print(void) {
        std::cout << "(" << x << ", " << y << ") ";
    }
};
```

```cpp
template <>
class Point<double> { // Specialized for double
private: double x, y;
public:
    Point(double m = 0, double n = 0) : x{m}, y{n} {}
    double getX(void) const { return x; }
    double getY(void) const { return y; }
    void print(void) { std::cout << std::scientific << "(" << x << ", " << y << ") "; }
};

int main(void) {
    Point<int> p{1, 2};         // An int type instance
    Point<double> q{2.5, 3.5}; // A double type instance
    p.print();
    q.print();
    return 0;
}
```

Generic Programming

- Template **metaprogramming** uses templates at compile time to do computation. For example:

```
template <int N>
struct factorial { static int const value = N * factorial<N - 1>::value; };
template <> struct factorial<1> { static int const value = 1; };
int main(void) { std::cout << "10! = "<< factorial<10>::value << std::endl; return 0; }
```

```
template <typename T> struct remove_const { using type = T; };
template <typename T> struct remove_const<const T> { using type = T; };
int main(void) {
  std::cout << std::is_same<int, remove_const<int>::type>::value;       // true
  std::cout << std::is_same<int, remove_const<const int>::type>::value; // true
  return 0;
}
```

```
template <uint8_t N> class count_ones { enum {
    bit7 = (N & 0x80) ? 1 : 0, bit6 = (N & 0x40) ? 1 : 0, bit5 = (N & 0x20) ? 1 : 0,
    bit4 = (N & 0x10) ? 1 : 0, bit3 = (N & 0x08) ? 1 : 0, bit2 = (N & 0x04) ? 1 : 0,
    bit1 = (N & 0x02) ? 1 : 0, bit0 = (N & 0x01) ? 1 : 0,
  }; public: enum { value = bit0 + bit1 + bit2 + bit3 + bit4 + bit5 + bit6 + bit7 };
};
int main(void) { std::cout << count_ones<255>::value << std::endl; return 0; }
```

```
template <int N, int M>
struct max { enum { value = (N > M) ? N : M }; };
int main(void) { std::cout << max<10, 20>::value; return 0; }
```

```
template <typename T, int exponent> struct power {
   static T base(T x) { return x * power<T, exponent - 1>::base(x); }
};

template <typename T> struct power<T, 0> {
  static T base(T x) { return 1; }
};

int main(void) {
  std::cout << power<int, 2>::base(3) << std::endl; /* 3 ^ 2 = 9 */
  return 0;
}
```

Generic Programming

❖ Templates provide a mechanism for **polymorphism** at compile time.

```
#include <iostream>

class Circle {
  double radius;
public:
  Circle(double r = 0) : radius{r} {}
  const char *getName(void) const { return "Circle"; }
  double getArea(void) const { return (3.1415 * radius *
radius); }
};

class Square {
  double length;
public:
  Square(double len = 0) : length{len} {}
  const char *getName(void) const { return "Square"; }
  double getArea(void) const { return (length * length); }
};
```

```
template <typename T>
void printArea(const T &shape)  {
    std::cout << shape.getName() << " Area = "
              << shape.getArea() << std::endl;
}

int main(void) {
  Circle c{10};
  printArea(c);

  Square s{10};
  printArea(s);

  return 0;
}
```

Generic Programming

- Possible to make variadic templates; i.e. templates with a variable number of parameters
- The **sizeof...** operator can be used to get the number of remaining elements a parameter pack contains

  E.g. sizeof...(args) or sizeof...(Types)

```cpp
template <typename... T>
void addOne(T const &...args) { print((args + 1)...); /* It can also be: print(args + 1 ...); */ }
```

```cpp
template <typename C, typename... Idx>
void printElems(C const &cont, Idx... idx) { print(cont[idx]...); }

template <std::size_t... Idx, typename T> void printIdx(T const &cont) { print(cont[Idx]...); }

int main(void) {
  std::vector<std::string> vec = {"A", "B", "C", "D", "E"};
  printElems(vec, 2, 0, 3); printIdx<2, 0, 3>(vec);

  return 0;
}
```

```cpp
#include <iostream>
template <typename T>
void print(T arg) { std::cout << arg << std::endl; }
template <typename T, typename... Types>
void print(T firstArg, Types... args) {
  print(firstArg); // print the first argument
  print(args...);  // print the remaining arguments
}
int main(void) {
  print(1); print(1, 2); print(1, 2.5f, 3.4);
  return 0;
}
```

Generic Programming

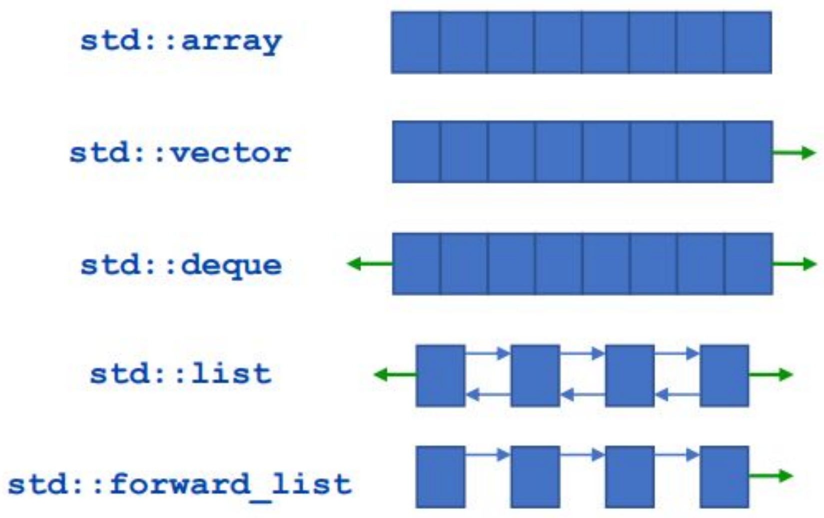
- Class template std::tuple is a fixed-size collection of heterogeneous values
- Possible to have template template parameters. E.g. `template <typename T, template <typename, typename> class U>`
- The typename and class keywords can exchangeably be used
  - typename is used to clarify that an identifier inside a template is a type. E.g. typename T::SubType* ptr;
  - When specifying a template template, the **class** keyword shall be used
  - Since C++17 both keywords are allowed to be used.

```
#include <tuple>
#include <iostream>

template <typename T, std::size_t SIZE> struct A { T array[SIZE]{}; };
template <typename T, std::size_t SIZE, template <typename, std::size_t> class U> struct B { U<T, SIZE> var; };
std::tuple<int, double, std::string> func(void) { return std::make_tuple(1, 2.5, "Hello"); }

int main(void) {
  B<int, 10, A> b; b.var.array[0] = 100; std::cout << b.var.array[0] << std::endl;
  std::tuple<int, double, std::string> tpl = func(); std::cout << std::get<2>(tpl) << std::endl;
  return 0;
}
```

Standard Template Library (STL)

- The containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queue, list and stack.
- There are three types of container: **Sequential**, **Associative** and **Container Adapters**
- **Sequential** containers implement data structures which can be accessed sequentially.
  - **Arrays** and **Vectors** are guaranteed to be stored in contiguous storage locations. This provides access through pointer like classic arrays.

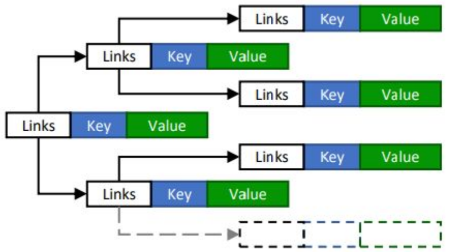
`array` (C++11)	static contiguous array (class template)
`vector`	dynamic contiguous array (class template)
`deque`	double-ended queue (class template)
`forward_list` (C++11)	singly-linked list (class template)
`list`	doubly-linked list (class template)



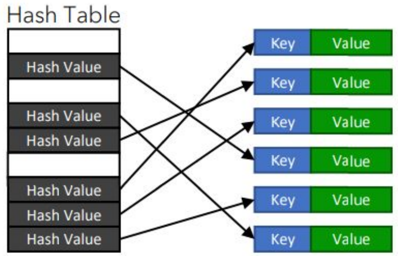
Standard Template Library (STL)

- The STL supports the following types of associative container:
  - map : collection of key-value pairs, keys are unique.
    - Elements are sorted by keys and stored in a balanced binary tree
  - unordered_map: collection of key-value pairs, hashed by keys, keys are unique
  - set: collection of unique keys, sorted by keys
  - unordered_set: collection of unique keys, hashed by keys
  - multimap and multiset: keys are not unique
    - They allow duplicate keys.

std::map



std::unordered_map



- Container adaptors: stack, queue and priority_queue
  - priority_queue: usually implemented in a tree data structure and the top is always the element with the highest priority.

Standard Template Library (STL)

❖ Some Examples

```
#include <queue>
#include <iostream>

int main(void) {
  std::priority_queue<int> q;
  q.push(100); q.push(50);
  q.push(1000); q.push(800);
  q.push(300);
  std::cout << "\nOrder values are inserted ... " << std::endl;
  std::cout << "100\t50\t1000\t800\t300" << std::endl;
  std::cout << "Priority queue values are ..." << std::endl;
  while (!q.empty()) {
    std::cout << q.top() << "\t";
    q.pop();
  }
  std::cout << std::endl;
  return 0;
}
```

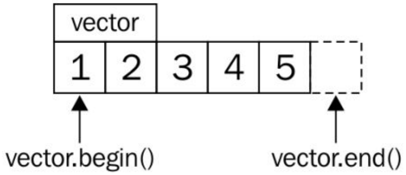
```
#include <map>
#include <iostream>

int main(void) {
  std::multimap<std::string, long> contacts = {
    {"Eva", 2232342343}, {"Linda", 3243435343},
    {"Markus", 6234324343}, {"Linda", 8932443241},
    {"Oliver", 5534327346}};
  auto pos = contacts.find("Linda");
  int count = contacts.count("Linda");

  int index = 0;
  while (pos != contacts.end()) {
    std::cout << "Mobile number of " << pos->first
                << " is " << pos->second << std::endl;
    pos++; index++;
    if (index == count) { break; }
  }
  return 0;
}
```

```
#include <map>
#include <iostream>

int main(void)
{
  std::map<std::string, long> contacts;
  contacts["Eva"] = 123456789;
  contacts["Lars"] = 523456289;
  contacts["Markus"] = 623856729;
  contacts["Linda"] = 993456789;

  auto pos = contacts.find("Linda");
  if (pos != contacts.end())
  {
    std::cout << pos->second << std::endl;
  }

  return 0;
}
```

Standard Template Library (STL)

❖ Iterators are a generalization of pointers that allow a C++ program to work with different data structures in a uniform manner.

Iterator category	Multi pass support	Defined operations
Input Iterator	multiple passes not supported	*it (read-access) ++it or it++
Forward Iterator	multiple passes supported	++it or it++
Bidirectional Iterator		--it or it--
Random Access Iterator		it+=n or it-=n
Contiguous Iterator		contiguous storage (like an array)




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

int main(void) {
  std::vector<int> vec{1, 2, 3, 4, 5};

  auto it{vec.begin()};
  while (it != vec.end()) {
    std::cout << *it << std::endl;
    ++it;
  }

  auto rit{vec.rbegin()};
  while (rit != vec.rend()) {
    std::cout << *rit << std::endl;
    ++rit;
  }

  return 0;
}
```

Standard Template Library (STL)

- STL file handling header includes:
  - std::ofstream: Output – write to file
  - std::ifstream: Inputs – read from file
  - std::fstream: Input/Output – read/write
  - This header is part of the Input/Output library.
- There are two types of file: text and binary
  - Use read() and write() function for binary files
  - Use the insertion/extraction operators for text files
- To use a file, first we shall open it.
  - There are different modes to open a file in.
- Opened files shall be closed before termination
  - Unwritten data in the output buffer will be written.

```
#include <deque>
#include <iostream>
#include <iterator>
#include <algorithm>

int main(void) {
  std::deque<int> d{10, 20, 30, 40, 50};
  std::cout << "Initial size of deque is " << d.size() << std::endl;
  d.push_back(60); d.push_front(5);
  std::cout << std::endl << "Size of deque after push back and front is "
          << d.size() << std::endl;
  std::copy(d.begin(), d.end(), std::ostream_iterator<int>(std::cout, "\t"));
  d.clear();
  std::cout << std::endl << "Size of deque after clearing all values is "
          << d.size() << std::endl;
  std::cout << std::endl << "Is the deque empty after clearing values ? "
          << (d.empty() ? "true" : "false") << std::endl << std::endl;

  return 0;
}
```

Standard Template Library (STL)

```cpp
#include <fstream>
#include <iostream>
struct data_t { int id; double value; };
constexpr int SIZE{2};
int main(void) {
  data_t data[SIZE]{{1, 2.45}, {2, 5.67}};
  std::fstream file{"file.txt", std::ios::in | std::ios::out | std::ios::trunc};
  if (!file.is_open()) {
    std::cout << "Failed to open file.txt!" << std::endl;  std::exit(1);
  }
  for (const auto &elem : data) {
    file << elem.id << ": " << elem.value << std::endl;
  }
  file.seekg(0, std::ios::beg);
  for (int i = 0; i < SIZE; i++) { data_t temp; file >> temp.id;
    file.seekg(2, std::ios::cur); file >> temp.value;
    std::cout << temp.id << ": " << temp.value << std::endl;
  }
  file.close();
  return 0;
}
```

```cpp
#include <fstream>
#include <iostream>
struct data_t { int id; double value; };
constexpr int SIZE{2};
int main(void) {
  data_t data[SIZE]{{1, 2.45}, {2, 5.67}};
  std::fstream file{"file.bin", std::ios::binary | std::ios::in | std::ios::out | std::ios::trunc};
  if (!file.is_open()) {
    std::cout << "Failed to open file.bin!" << std::endl; std::exit(1);
  }
  for (const auto &elem : data) {
    file.write(reinterpret_cast<char *>(const_cast<data_t *>(&elem)), sizeof(data_t));
  }
  file.seekg(0, std::ios::beg);
  for (int i = 0; i < SIZE; i++) {
    data_t temp; file.read(reinterpret_cast<char *>(&temp), sizeof(data_t));
    std::cout << temp.id << ": " << temp.value << std::endl;
  }
  file.close();
  return 0;
}
```

C++ Language

- Some useful links
  - C++ Reference
  - Standard C++ Library Reference
  - C++ Tutorial
  - C++ Language
  - C++ Tutorial
  - C++ Full Course For Beginners
  - C++ Tutorial for Beginners - Full Course
  - C++ Tutorial 2021
  - C++ Programming Tutorials Playlist