模板与概念

概述

- 目的：可重用的代码
- 函数可用于编写不依赖特定值的算法，从而可重用不同的值
- 模板不仅允许参数化值，还允许参数化类型
  - 例：vector

主要内容

- 类模板
- 函数模板
- 概念

类模板

- 将类的一些数据成员的类型，方法的返回类型，方法的参数类型指定为参数

实例：通用的棋盘类

- 要求：可用于象棋棋盘，跳棋棋盘，井字棋棋盘等二维棋盘。能够保存象棋棋子，跳棋棋子，井字棋棋子等任何类型的棋子

不使用模板

- 采用多态，棋盘保存通用的GamePiece对象

```
export class GamePiece {
public:
        virtual ~GamePiece() = default;
        virtual std::unique_ptr<GamePiece> clone() const = 0;
};
```

- 每种游戏的棋子继承GamePiece

```
class ChessPiece : public GamePiece {
public:
        std::unique_ptr<GamePiece> clone() const override
        {
                // Call the copy constructor to copy this instance
                return std::make_unique<ChessPiece>(*this);
        }
};
```

类的定义

```
export class GameBoard  {
public:    explicit GameBoard(size_t width = DefaultWidth, size_t height = DefaultHeight);
           GameBoard(const GameBoard& src);   // copy constructor
           virtual ~GameBoard() = default;    // virtual defaulted destructor
           GameBoard& operator=(const GameBoard& rhs); // assignment operator
           // Explicitly default a move constructor and move assignment operator.
           GameBoard(GameBoard&& src) = default;
           GameBoard& operator=(GameBoard&& src) = default;
           std::unique_ptr<GamePiece>& at(size_t x, size_t y);  //两个版本的at，返回的都是引用，而不是对象（的拷贝）
           const std::unique_ptr<GamePiece>& at(size_t x, size_t y) const;
           size_t getHeight() const { return m_height; }                size_t getWidth() const { return m_width; }
           static const size_t DefaultWidth{ 10 };              static const size_t DefaultHeight{ 10 };
           void swap(GameBoard& other) noexcept;
private:   void verifyCoordinate(size_t x, size_t y) const;
           std::vector<std::vector<std::unique_ptr<GamePiece>>> m_cells;  //存放棋子的棋盘，可看作一个二维数组
           size_t m_width { 0 }, m_height { 0 };
};
export void swap(GameBoard& first, GameBoard& second) noexcept;
```

实现

```
GameBoard::GameBoard(size_t width, size_t height) : m_width{ width }, m_height{ height } {
        m_cells.resize(m_width);
        for (auto& column : m_cells)
                column.resize(m_height);
}
GameBoard::GameBoard(const GameBoard& src): GameBoard{ src.m_width, src.m_height } {
        // The ctor-initializer of this constructor delegates first to the
        // non-copy constructor to allocate the proper amount of memory.
        // The next step is to copy the data.
        for (size_t i{ 0 }; i < m_width; i++)
                for (size_t j{ 0 }; j < m_height; j++)
                        if (src.m_cells[i][j])    m_cells[i][j] = src.m_cells[i][j]->clone();
}
void GameBoard::verifyCoordinate(size_t x, size_t y) const  {
        if (x >= m_width)   throw std::out_of_range { std::format("{} must be less than {}.", x, m_width) };
        if (y >= m_height) throw std::out_of_range { std::format("{} must be less than {}.", y, m_height) };
}
```

实现

```
void GameBoard::swap(GameBoard& other) noexcept  {
        std::swap(m_width, other.m_width);              std::swap(m_height, other.m_height);
        std::swap(m_cells, other.m_cells);
}
void swap(GameBoard& first, GameBoard& second) noexcept  {
        first.swap(second);
}
GameBoard& GameBoard::operator=(const GameBoard& rhs)  {  // Copy-and-swap idiom
        GameBoard temp{ rhs }; // Do all the work in a temporary instance
        swap(temp);   // Commit the work with only non-throwing operations
        return *this;
}
const unique_ptr<GamePiece>& GameBoard::at(size_t x, size_t y) const  {
        verifyCoordinate(x, y);                 return m_cells[x][y];
}
unique_ptr<GamePiece>& GameBoard::at(size_t x, size_t y)  {
        return const_cast<unique_ptr<GamePiece>&>(as_const(*this).at(x, y));
}
```

缺点

- 无法使用GameBoard按值存储元素，只能存储指针
- 当使用at访问某个格子时，得到的是unique_ptr<GamePiece>，需要转换为ChessPiece才能使用ChessPiece的功能，而这个转换又需要记住格子中棋子的类型才能做正确的类型转换
- 可以在一种棋盘存储不同类型的棋子
- 不能存储int，float等基本类型，只能存储GamePiece的派生类

模板化的Grid类

```
export  template <typename T>
class Grid  {
public:     explicit Grid(size_t width = DefaultWidth, size_t height = DefaultHeight);
            virtual ~Grid() = default;
            // Explicitly default a copy constructor and copy assignment operator.
            Grid(const Grid& src) = default;
            Grid& operator=(const Grid& rhs) = default;
            // Explicitly default a move constructor and move assignment operator.
            Grid(Grid&& src) = default;
            Grid& operator=(Grid&& rhs) = default;
            std::optional<T>& at(size_t x, size_t y);
            const std::optional<T>& at(size_t x, size_t y) const;
            size_t getHeight() const { return m_height; }
            size_t getWidth() const { return m_width; }
            static const size_t DefaultWidth{ 10 };
            static const size_t DefaultHeight{ 10 };
private:    void verifyCoordinate(size_t x, size_t y) const;
            std::vector<std::vector<std::optional<T>>> m_cells;
            size_t m_width { 0 }, m_height { 0 };
};
```

定义基于类型T的类模板
由于历史的原因，也可用class替代typename
T是模板类型参数

rhs的类型不再是const GameBoard&，而是const Grid&，编译器将其解释为Grid<T>，可以直接写为Grid<T>& operator=(const Grid<T>& rhs) = default;

at返回optional<T>&

m_cells存储真正的对象而不是指针。通过optional<T>，即可以是T类型的值，也可以为空，

模板化的Grid类的实现

```
template <typename T>  //每个方法定义前需要这一行，说明这是一个函数模板
Grid<T>::Grid(size_t width, size_t height) : m_width{ width }, m_height{ height } {  //注意开头的Grid<T>::
        m_cells.resize(m_width);       for (auto& column : m_cells)   column.resize(m_height);
}
template <typename T>
void Grid<T>::verifyCoordinate(size_t x, size_t y) const  {
        if (x >= m_width)  throw std::out_of_range{ std::format("{} must be less than {}.", x, m_width) };
        if (y >= m_height)   throw std::out_of_range{ std::format("{} must be less than {}.", y, m_height) };
}
template <typename T>
const std::optional<T>& Grid<T>::at(size_t x, size_t y) const  {
        verifyCoordinate(x, y);          return m_cells[x][y];
}
template <typename T>
std::optional<T>& Grid<T>::at(size_t x, size_t y)  {
        return const_cast<std::optional<T>&>(std::as_const(*this).at(x, y));  //注意返回值类型的写法
}
```

使用Grid模板

```
//以Grid对象为形参的函数，必须在Grid类型中指定保存在Grid中的元素类型，或者编写函数模板
void processIntGrid(Grid<int>& /*grid*/) { ... }
int main() {
        Grid<int> myIntGrid; // 模板实例化 Declares a grid that stores ints, using default arguments for the constructor.
        Grid<double> myDoubleGrid{ 11, 11 }; // Declares an 11x11 Grid of doubles.
        myIntGrid.at(0, 0) = 10;
        int x{ myIntGrid.at(0, 0).value_or(0) };//at返回optional引用，如果包含值，value_or返回该值，否则返回value_or的实参
        Grid<int> grid2{ myIntGrid };  // Copy constructor
        Grid<int> anotherIntGrid;  //可为模板类指定简单的名称，using IntGrid=Grid<int>
        anotherIntGrid = grid2;      // Assignment operator
        //Grid test;   // 无法编译，需要提供模板参数
        //Grid<> test; // 无法编译，模板参数太少
        processIntGrid(myIntGrid);
        //以下展示保存各种类型的数据，但是int型的Grid不能存储SpreadsheetCell等其它类型
        Grid<SpreadsheetCell> mySpreadsheet;   SpreadsheetCell myCell{ 1.234 };          mySpreadsheet.at(3, 4) = myCell;
        Grid<const char*> myStringGrid;        myStringGrid.at(2, 2) = "hello";
        Grid<vector<int>> gridOfVectors;       vector<int> myVector{ 1, 2, 3, 4 };       gridOfVectors.at(5, 6) = myVector;
        auto myGridOnFreeStore { make_unique<Grid<int>>(2, 2) }; // 动态分配
        myGridOnFreeStore->at(0, 0) = 10;      int x2 { myGridOnFreeStore->at(0, 0).value_or(0) };
}
```

编译器处理模板的原理

- 遇到模板方法定义时，会进行一些简单的语法检查，但是不编译
- 遇到实例化的模板时，为相应的类型生成一个模板类，例如Grid<int>，会将类模板定义中的每一个类型参数替换为int，得到一个int版本的Grid模板类
- 如果程序中没有实例化，则不会编译类模板中任何方法的定义
- 选择性实例化：编译器总是为类模板的所有虚方法生成代码，但只会为那些实际被使用的非虚方法生成代码。未编译部分可能存在的语法错误不会被发现。

  Grid<int> myIntGrid;
  myIntGrid.at(0,0)=10;
  若程序只有以上两行，则只为int版的Grid生成无参构造函数，析构函数和非常量的at方法的代码
- 可通过显式的模板实例化强制编译器为所有的类方法生成代码，包括虚方法和非虚方法。例如template class Grid<int>;
- 在编写与类型无关的代码时，必须对这些类型有一些假设，例如Grid模板假设元素类型T是可析构，可复制/移动构造，可复制/移动赋值的
- 概念允许编写编译器可以解释和验证的模板参数要求

如何组织文件

- 对于类模板，编译器必须可以从使用它们的任何源文件中获取类模板定义和方法定义
  1）将方法定义直接放在模板定义所在的模块接口文件中，使用模板的文件导入此模块时编译器能够访问所需的所有代码
  2）将类模板方法的定义和类模板的定义放在不同分区

// Grid.cppm

```
export module grid;
export import :definition;
export import :implementation;
```

//GridDefinition.cppm

```
module;
export module grid:definition;
export
template <typename T>
class Grid {...};
```

//GridImplementation.cppm

```
module;
export module grid:implementation;
import :definition;

export
template <typename T>
Grid<T>::Grid(size_t width, size_t height)
            : m_width{ width } , m_height{ height }
{...}
...
```

非类型的模板参数

- 模板参数可以有任意多个，可以不是类型，可以有默认值
- 非类型的模板参数：只能是整数类型(int, char, long等)，枚举，指针，引用，std::nullptr_t，auto，auto&和auto*，C++20虽然可以使用浮点甚至类类型，但有很多限制
- 使用非类型模板参数的好处是在编译代码前就知道这些参数的值，而编译器为方法生成代码时会在编译前替换模板参数

//Grid.cppm

```
export module grid;
export
template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid  {
public:
        Grid() = default;
        virtual ~Grid() = default;
        // Explicitly default a copy constructor and copy assignment operator.
        Grid(const Grid& src) = default;
        Grid& operator=(const Grid& rhs) = default;
        std::optional<T>& at(size_t x, size_t y);
        const std::optional<T>& at(size_t x, size_t y) const;
        size_t getHeight() const { return HEIGHT; }
        size_t getWidth() const { return WIDTH; }
private:
        void verifyCoordinate(size_t x, size_t y) const;
        std::optional<T> m_cells[WIDTH][HEIGHT]; //可以使用普通数组而不是动态分配
};

template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::verifyCoordinate(size_t x, size_t y) const{  ...  }
.....
```

//GridTest.cpp

```
import grid;
constexpr size_t getHeight()  {
        return 10;
}

int main()  {
        Grid<int, 10, 10> myGrid;
        Grid<int, 10, 10> anotherGrid;
        myGrid.at(2, 3) = 42;
        anotherGrid = myGrid;
        cout << anotherGrid.at(2, 3).value_or(0);
        Grid<double, 2, getHeight()>    myDoubleGrid;
}
```

缺点：
不能通过非常量的常数指定高度和宽度，
Grid<int,10,height>testGrid就无法编译
不同参数值意味着不同的类型，Grid<int,5,5>和Grid<int,5,6>是两种类类型，相互之间不能直接赋值等操作（后面通过方法模板解决）

模板参数的默认值

- 默认值在类定义时提供，不需要在方法定义中指定
- 使用规则与函数相同，可以从右往左提供参数的默认值

//Grid.cppm

```
export module grid;

export
template <typename T = int, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid {...};

template <typename T, size_t WIDTH, size_t HEIGHT>
const std::optional<T>& Grid<T, WIDTH, HEIGHT>::at(size_t x, size_t y) const
{
        verifyCoordinate(x, y);
        return m_cells[x][y];
}
```

//GridTest.cpp

```
import grid;
int main()
{
        Grid<> myIntGrid;
        Grid<int> myGrid;
        Grid<int, 5> anotherGrid;
        Grid<int, 5, 5> aFourthGrid;
        //Grid myDefaultGrid;  //错误
}
```

类模板的实参推导

- 从传递给类模板构造函数的实参推导出模板参数
- C++17之前unique_ptr和shared_ptr禁用类型推导，c++17及以后虽然可以推导，但是更推荐使用make_unique()和make_shared()两个函数，这样更安全高效
- 用户可以编写如何推导模板参数的规则

```
template <typename T>
class SpreadsheetCell  {
public:          SpreadsheetCell(T t) : m_content{ move(t) } { }
                 const T& getContent() const { return m_content; }
private:         T m_content;
};

SpreadsheetCell(const char*) -> SpreadsheetCell<std::string>; // 用户为const char*特化提供推导指南

int main(){
        string myString{ "Hello World!" };
        SpreadsheetCell cell{ myString };
        SpreadsheetCell cell2{ “test” };  //推导为SpreadsheetCell<std::string>
}
```

方法模板

- 可以模板化普通类以及类模板中的单个方法，虚方法和析构函数除外
- 例如Grid<int>和Grid<double>是两种不同的模板类，以下操作无法编译
  myDoubleGrid=myIntGrid;
  Grid<double> newDoubleGrid{myIntGrid};
- 因为Grid的拷贝构造函数和赋值运算符的定义为
  Grid(const Grid&);
  Grid& operator=(const Grid&);
- 等价于
  Grid(const Grid<T>&);
  Grid<T> operator=(const Grid<T>&);

```
export template <typename T>
class Grid  {
public:     explicit Grid(size_t width = DefaultWidth, size_t height = DefaultHeight);   virtual ~Grid() = default;
            // Explicitly default a copy constructor and copy assignment operator.
            Grid(const Grid& src) = default;Grid& operator=(const Grid& rhs) = default;
            // Explicitly default a move constructor and move assignment operator.
            Grid(Grid&& src) = default;       Grid& operator=(Grid&& rhs) = default;
            template <typename E>  Grid(const Grid<E>& src); //不能删除初始的拷贝构造函数和赋值运算符
            template <typename E>  Grid& operator=(const Grid<E>& rhs); //如果E=T，则调用初始版本
            void swap(Grid& other) noexcept;              ...
};

template <typename T>  //双重模板化，类在类型T上被模板化，而新版拷贝构造函数在不同的类型E上模板化
template <typename E>  //不能写成template<typename T, typename E>
Grid<T>::Grid(const Grid<E>& src): Grid{ src.getWidth(), src.getHeight() } { //形参为Grid<E>&，返回类型为Grid<T>&
            // The ctor-initializer of this constructor delegates first to the
            // non-copy constructor to allocate the proper amount of memory.
            // The next step is to copy the data.
            for (size_t i{ 0 }; i < m_width; i++)
                        for (size_t j{ 0 }; j < m_height; j++)
                                    m_cells[i][j] = src.at(i, j);
}

template <typename T>
void Grid<T>::swap(Grid& other) noexcept  {
            std::swap(m_width, other.m_width);
            std::swap(m_height, other.m_height);
            std::swap(m_cells, other.m_cells);       }
```

```
template <typename T>
template <typename E>
Grid<T>& Grid<T>::operator=(const Grid<E>& rhs)  {
            // Copy-and-swap idiom
            Grid<T> temp{ rhs }; // Do all the work in a temporary instance.
            //rhs是Grid<E>&，通过新版拷贝构造函数得到Grid<T>的temp
            swap(temp);
            return *this;
}
```

带有非类型参数的方法模板

- 类似
- 可以将不同高度和宽度的Grid进行拷贝构造/赋值，不一定要把源对象完美复制到目标对象，一方面可以从源对象中只复制那些能够被放入目标对象的元素，另一方面，如果源对象在某个维度上小于目标对象，可以用默认值进行填充

```
export  template <typename T, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid {
public:     Grid() = default;       virtual ~Grid() = default;
            // Explicitly default a copy constructor and copy assignment operator.
            Grid(const Grid& src) = default;              Grid& operator=(const Grid& rhs) = default;
            template <typename E, size_t WIDTH2, size_t HEIGHT2>        Grid(const Grid<E, WIDTH2, HEIGHT2>& src);
            template <typename E, size_t WIDTH2, size_t HEIGHT2>        Grid& operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs);
            void swap(Grid& other) noexcept;.           ......
};

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid<T, WIDTH, HEIGHT>::Grid(const Grid<E, WIDTH2, HEIGHT2>& src)  {
            for (size_t i{ 0 }; i < WIDTH; i++)   //注意循环的范围
                        for (size_t j{ 0 }; j < HEIGHT; j++)
                                    if (i < WIDTH2 && j < HEIGHT2)    m_cells[i][j] = src.at(i, j);
                                    else                        m_cells[i][j].reset();
}

template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::swap(Grid& other) noexcept  {  std::swap(m_cells, other.m_cells);  }

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid<T, WIDTH, HEIGHT>& Grid<T, WIDTH, HEIGHT>::operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs)  {
            Grid<T, WIDTH, HEIGHT> temp{ rhs }; // Do all the work in a temporary instance.
            swap(temp); // Commit the work with only non-throwing operations.
            return *this;     }
```

类模板的特化

- 给类模板的特定类型提供不同的实现
- Grid<const char*>将使用vector<vector<optional<const char*>>>存储元素，相应的拷贝构造和赋值执行const char*的浅复制而不是深复制
- 编写类模板的特化时，必须指明这是一个模板以及为哪种特定类型编写这个模板
- 为const char*特化的Grid实现中，初始的Grid模板放置在main模块接口分区中，特化部分放置在string模块接口分区

```
//Grid.cppm 模块接口文件
export module grid;
export import :main;
export import :string;
```

- 特化的好处是可以对用户隐藏，用户创建Grid<int>时自动从初始的Grid模板生成代码，用户创建Grid<const char*>时自动使用特化版本
- 特化与派生不同，不继承任何代码，需要重新编写类的所有实现，不要求提供具有相同的名称或行为的方法。每个方法定义之前不必写template<>
- 不要滥用特化

//GridString.cppm

```
export module grid:string;
// When the template specialization is used, the original template must be visible too.
import :main; //原来的Grid类模板假设在一个名为main的模块接口文件

export template <> //告诉编译器这是Grid的const char*特化版本，不指定任何类型参数
class Grid<const char*> {
public:    explicit Grid(size_t width = DefaultWidth, size_t height = DefaultHeight);
           virtual ~Grid() = default;
           // Explicitly default a copy constructor and copy assignment operator.
           Grid(const Grid& src) = default;
           Grid& operator=(const Grid& rhs) = default;
           // Explicitly default a move constructor and move assignment operator.
           Grid(Grid&& src) = default;
           Grid& operator=(Grid&& rhs) = default;
           std::optional<std::string>& at(size_t x, size_t y);
           const std::optional<std::string>& at(size_t x, size_t y) const;
           size_t getHeight() const { return m_height; }     size_t getWidth() const { return m_width; }
           static const size_t DefaultWidth{ 10 };           static const size_t DefaultHeight{ 10 };
private:   void verifyCoordinate(size_t x, size_t y) const;
           std::vector<std::vector<std::optional<std::string>>> m_cells;
           size_t m_width { 0 }, m_height { 0 };
};
```

```
//不需要template<>
Grid<const char*>::Grid(size_t width, size_t height) : m_width{ width }, m_height{ height } {
	m_cells.resize(m_width);
	for (auto& column : m_cells)		column.resize(m_height);
}

void Grid<const char*>::verifyCoordinate(size_t x, size_t y) const  {
	if (x >= m_width)   throw std::out_of_range { std::format("{} must be less than {}.", x, m_width) };
	if (y >= m_height) throw std::out_of_range { std::format("{} must be less than {}.", y, m_height) };
}

const std::optional<std::string>& Grid<const char*>::at(size_t x, size_t y) const {
	verifyCoordinate(x, y);
	return m_cells[x][y];
}

std::optional<std::string>& Grid<const char*>::at(size_t x, size_t y) {
	return const_cast<std::optional<std::string>&>(std::as_const(*this).at(x, y));
}
```

简单的应用

```
int main()
{
	Grid<int> myIntGrid;		// Uses original Grid template.
	Grid<const char*> stringGrid1{ 2, 2 };// Uses const char* specialization.

	const char* dummy{ "dummy" };
	stringGrid1.at(0, 0) = "hello";
	stringGrid1.at(0, 1) = dummy;
	stringGrid1.at(1, 0) = dummy;
	stringGrid1.at(1, 1) = "there";

	Grid<const char*> stringGrid2{ stringGrid1 };

	cout << stringGrid1.at(0, 1).value_or("") << endl;
	cout << stringGrid2.at(0, 1).value_or("") << endl;
}
```

从类模板派生

- 可以从模板类派生得到一个新的模板类，也可以派生得到一个特定实例
- 语法与普通继承一样

//GameBoard.cppm

```
import grid;
export template <typename T>
class GameBoard : public Grid<T> {
public:    explicit GameBoard(size_t width = Grid<T>::DefaultWidth, size_t height = Grid<T>::DefaultHeight);
           void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest);
};
template <typename T>
GameBoard<T>::GameBoard(size_t width, size_t height) : Grid<T>{ width, height } {}
template <typename T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest) {
           //使用this或Grid<T>::来指明基类模板中的数据成员和方法
           Grid<T>::at(xDest, yDest) = std::move(Grid<T>::at(xSrc, ySrc));
           Grid<T>::at(xSrc, ySrc).reset(); // Reset source cell
           // this->at(xDest, yDest) = std::move(this->at(xSrc, ySrc));
           // this->at(xSrc, ySrc).reset();
}
```

//GameBoardTest.cpp

```
int main() {
           GameBoard<ChessPiece> chessBoard{ 8, 8 };   ChessPiece pawn;
           chessBoard.at(0, 0) = pawn; chessBoard.move(0, 0, 0, 1);
}
```

继承与特化

- 通过继承扩展实现和使用多态，通过特化自定义实现特定类型

继承
派生类包含基类的所有成员和方法
派生类名与基类名不同
派生类的对象可以代替基类的对象

特化
必须重写所有的代码
特化的名称必须和原始名称一致
每个实例化都是一种不同的类型

模板别名

- 假设有如下类模板
  template<typename T1, typename T2>
  class MyTemplateClass {...};
- 可以给定两个类型参数来定义别名
  using OtherName=MyTemplateClass<int, double>;//可以用typedef
- 还可以指定部分类型，这又称为别名模板
  template<typename T1>
  using otherName=MyTemplateClass<T1, double>; //此时不能用typedef

主要内容

- 类模板
- 函数模板
- 概念

函数模板

- 函数也可以有模板
- 例：数组的顺序查找

```
static const size_t NOT_FOUND{ static_cast<size_t>(-1) };

template <typename T>
size_t Find(const T& value, const T* arr, size_t size)
{
	for (size_t i{ 0 }; i < size; i++) {
		if (arr[i] == value) {
			return i; // Found it; return the index.
		}
	}
	return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

```
int main()  {
        int myInt{ 3 }, intArray[]{ 1, 2, 3, 4 };
        const size_t sizeIntArray{ size(intArray) };
        size_t res;
        res = Find(myInt, intArray, sizeIntArray);        // calls Find<int> by deduction.
        res = Find<int>(myInt, intArray, sizeIntArray);   // calls Find<int> explicitly.
        if (res != NOT_FOUND) { cout << res << endl; }
        else { cout << "Not found" << endl; }
        double myDouble{ 5.6 }, doubleArray[]{ 1.2, 3.4, 5.7, 7.5 };
        const size_t sizeDoubleArray{ size(doubleArray) };
        res = Find(myDouble, doubleArray, sizeDoubleArray);         // calls Find<double> by deduction.
        res = Find<double>(myDouble, doubleArray, sizeDoubleArray); // calls Find<double> explicitly.
        if (res != NOT_FOUND) { cout << res << endl; }
        else { cout << "Not found" << endl; }
        //res = Find(myInt, doubleArray, sizeDoubleArray);        // DOES NOT COMPILE! Arguments are different types.
        res = Find<double>(myInt, doubleArray, sizeDoubleArray);  // calls Find<double> explicitly, even with myInt.
        SpreadsheetCell cell1{ 10 };
        SpreadsheetCell cellArray[]{ SpreadsheetCell{ 4 }, SpreadsheetCell{ 10 } };
        const size_t sizeCellArray{ size(cellArray) };
        res = Find(cell1, cellArray, sizeCellArray);                    // calls Find<SpreadsheetCell> by deduction.
        res = Find<SpreadsheetCell>(cell1, cellArray, sizeCellArray); // calls Find<SpreadsheetCell> explicitly.
}
```

函数模板

- 可以使用非类型参数

```
template <typename T, size_t N>
size_t Find(const T& value, const T(&arr)[N])
{
    return Find(value, arr, N);
}
```

- 参数也可以使用默认值
- 函数模板的定义（不仅是原型）必须对所有使用它们的源文件可用，因此需要将其放入模板接口文件并导出

函数模板的重载

- 使用非模板函数重载函数模板

```
size_t Find(const char* value, const char** arr, size_t size)  {
        for (size_t i{ 0 }; i < size; i++)
                if (strcmp(arr[i], value) == 0)         return i; // Found it; return the index.
        return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

- 使用示例

```
int main()  {
        const char* word{ “two” };  //此处不用string，这样原来模板中使用==判别相等就不适用了
        const char* words[]{ "one", "two", "three", "four" };
        const size_t sizeWords{ size(words) };
        size_t res{ Find(word, words, sizeWords) }; // Calls non-template function.
        if (res != NOT_FOUND) { cout << res << endl; }
        else { cout << "Not found" << endl; }
        res = Find<const char*>(word, words, sizeWords); // Calls template with T=const char*.
}
```

类模板的友元函数模板

- 将两个Grid相加规模取min，只有两个格子都有实际值时才真正相加

```
export module grid;
export template <typename T> class Grid; // Forward declare Grid template.
export template<typename T>  Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs);
export template <typename T>
class Grid { public:  friend Grid operator+<T>(const Grid& lhs, const Grid& rhs);    //operator+<T>模板是Grid<T>的友元
.....};
export template <typename T> Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs) {
    size_t minWidth{ std::min(lhs.getWidth(), rhs.getWidth()) };  size_t minHeight{ std::min(lhs.getHeight(), rhs.getHeight()) };
    Grid<T> result{ minWidth, minHeight };
    for (size_t y{ 0 }; y < minHeight; ++y)
            for (size_t x{ 0 }; x < minWidth; ++x) {
                    const auto& leftElement{ lhs.m_cells[x][y] };      const auto& rightElement{ rhs.m_cells[x][y] };
                    if (leftElement.has_value() && rightElement.has_value())
                            result.at(x, y) = leftElement.value() + rightElement.value(); //需要格子中的元素支持加法
            }
            return result;
}
```

模板参数推导

- 编译器根据传递给函数模板的实参来推导模板参数的类型，对无法推导的模板参数，则显式指定
- 如下add函数模板需要3个模板参数

```
template<typename RetType, typename T1, typename T2>
RetType addA(const T1& t1, const T2& t2) { return t1 + t2; }
```

- 使用这个模板时，可以指定所有3个参数：

```
auto result1{ addA<long long, int, int>(1, 2) };
```

- 由于模板参数T1和T2可以通过实参进行推导，所以可仅指定返回值的类型

```
auto result2{ addA<long long>(1, 2) };
```

- 但是这仅在要推导的参数位于参数列表最后时才可行：

```
template<typename T1, typename RetType, typename T2>
RetType addB(const T1& t1, const T2& t2) { return t1 + t2; }
auto result1{ addB<long long, int, int>(1, 2) }; //错误，需要显式指定T1
```

- 可提供返回类型的默认值

```
template <typename RetType = long long, typename T1, typename T2>
RetType addC(const T1& t1, const T2& t2) { return t1 + t2; }
auto result{ addC(1, 2) };
```

函数模板的返回类型

- 由编译器推导返回值的类型

```
// From C++14: using automatic function return type deduction
//auto会去掉引用和const，除非写成const auto&。此例的加法生成新对象，所以没有影响，但可能不适用其它场景
template<typename T1, typename T2>
auto add1(const T1& t1, const T2& t2) {          return t1 + t2;    }

// From C++14: using decltype(auto)
//decltype不会去掉引用和const
template<typename T1, typename T2>
decltype(auto) add2(const T1& t1, const T2& t2)  {        return t1 + t2;     }

// C++14之前错误做法，开头就使用t1和t2，但是此时不知道t1和t2是什么
template<typename T1, typename T2>
decltype(t1+t2) add3(const T1& t1, const T2& t2)  {       return t1 + t2;     }

// C++14之前，使用替换函数语法，后置返回类型，因此在解析时参数名称，参数类型以及t1+t2类型已知
template<typename T1, typename T2>
auto add4(const T1& t1, const T2& t2) -> decltype(t1 + t2)  {       return t1 + t2;    }
```

简化函数模板

- 前一页的做法：
  template<typename T1, typename T2>
  decltype(auto) add2(const T1& t1, const T2& t2) {　　return t1 + t2;　}
- 简化函数模板：
  decltype(auto) add(const auto& t1, const auto& t2) {　return t1 + t2;　}
- 此时没有templatel来指定模板参数，但是每个被指定为auto的函数参数都成为模板类型参数
- 注意，每个被指定为auto的参数都会具有不同的模板类型参数，下例就无法做类似简化
  template<typename T>
  decltype(auto) add2(const T& t1, const T& t2) {return t1 + t2;　}
- 另外不能在函数模板的实现中显式使用导出的类型，因为这些自动导出的类型没有名称。如果需要这样做，一种方法是使用普通的函数模板语法，另一种是使用decltype确定

变量模板

- 下面是pi值的可变模板，从而用户可以得到所请求类型的中可表示的pi值

```
template <typename T>
constexpr T pi{ T{ 3.14159265358979323846264338327950288 4 } };

int main()
{
        float piFloat{ pi<float> };
        auto piLongDouble{ pi<long double> };
}
```

- C++20引入<numbers>，其中定义了一组常用的数学常量，其中pi为std::numbers::pi

主要内容

- 类模板
- 函数模板
- 概念

概念

- 背景：为模板提供错误参数时，编译器给出数百行错误信息
- 概念（Concepts） 是模板编程的重大改进，它通过约束模板参数（类型参数和非类型参数），显著提升了代码的可读性、错误信息清晰性和泛型编程的灵活性
- 编写概念是在为语义建模，而不仅仅是语法建模
- 概念定义的泛型语法如下

  template <parameter-list>

  concept concept-name=constraints-expression;
- 虽然以template开始，但是从不会被实例化，constraints-expression可以是任意的（编译时可计算的）常量表达式，必须产生一个布尔值，运行时不会被计算。用以表示模板参数必须满足的条件
- 概念表达式的语法为：concept-name<argument-list>，其结果为真或假，如果为真则表示使用给定的模板实参为概念建模

约束表达式

- 最简单的约束表达式是计算结果（精确且无类型转换）为布尔值的常量表达式
  template <typename T>
  concept C=sizeof(T)==4;
- require表达式是一种新的常量表达式类型，用以明确模板参数的要求。
  requires (parameter-list) { requirements; }
- parameter-list为可选参数，每个requirements以分号结尾
- 4种类型的requirement：简单，类型，复合，嵌套

简单requirement

- 是一个任意的表达式语句，不允许使用变量声明，循环，条件语句等，该表达式语句永远不会被计算，编译器只是验证是否通过编译
- 下例概念定义指定类型T必须是可递增的，支持前缀++和后缀++运算符

```
template <typename T>
concept Incrementable = requires(T x) { x++; ++x; }
```

- require表达式的参数列表用于引入位于require表达式主体中的命名变量，且require表达式的主体不能有常规变量的声明

类型requirement

- 用于验证是否是一种特定类型
- 下例要求T有value_type成员

```
template <typename T>
concept C=requires { typename T::value_type; };
```

- 类型需求可以用来验证某个模板是否可以使用给定的类型进行实例化

```
template <typename T>
concept C = requires { typename SomeTemplate<T>; };
```

复合requirement

- 用于验证某些东西不会抛出任何异常和/或验证某个方法是否返回某个类型，下面语法格式中noexcept和->type-constraint是可选

  { expression } noexcept -> type-constraint;
- 下例验证给定类型是否具有标记为noexcept的swap方法

  template <typename T>

  concept C=requires(T x, T y) { { x.swap(y) } noexcept; };
- 下例验证给定类型是否具有一个size方法，且该方法的返回类型可转换为size_t

  template <typename T>

  concept C=requires(const T x) { { x.size() } -> convertible_o<size_t>; };
- std::convertible_to<From, to>是标准库在<concepts>中定义的概念，具有两个类型参数，->左边的表达式的类型自动作为第一个类型参数传递给convertible，此时只需指定To对应的实参（此例中为size_t）
- 下例要求类型T可比较

  template <typename T>

  concept Comparable=requires(const T a, const T b) {

  { a==b } -> convertible_o<bool>;

  { a<b } -> convertible_o<bool>;

  //以及其它比较运算

  };

嵌套requirement

- 下例要求类型大小为4个字节，支持前缀和后缀的++与--

```
template <typename T>
concept C=requires (T t) {
    requires sizeof(t) == 4;
    ++t; --t; t++; t--;
};
```

概念表达式的组合

- 使用&&和||对已有概念进行组合
- 设已经有一个Incrementable和一个Decrementable的概念，定义一个新的概念，要求一个类型既可以递增又可以递减

```
template <typename T>
concept IncrementableAndDecrementable=Incrementable<T> && Decrementable<T>
```

已有的概念

- concepts库
  - 核心语言概念：same_as, derived_from, convertible_to, integral, floating_point, copy_constructibledeng
  - 比较概念：equality-comparable, totaly_ordered等
  - 对象概念：movable, copyable等
  - 可调用的概念：invocable, predicate等
- iterator库：random_access_iterator, forward_iterator等
- 尽量了解已有的概念，直接或者组合使用这些标准功能

auto的类型约束

- 类型约束可以用于：约束自动推导定义的变量，约束使用函数返回类型推导出的返回类型，约束简化函数模板和泛型lamda表达式中的参数
- 下面的代码可以正常编译，因为推导出的类型是 int，而 int 是增量概念的模型
  Incrementable auto value1 {1};
- 下面的代码会导致编译错误。类型被推断为std::string（注意后缀 s 是一个用户定义的字面量操作符（定义在 std::string_literals 命名空间中），它会将 const char* 转换为 std::string），而字符串并不是能够Incrementable的类型
  Incrementable auto value {“abc”s};

函数模板的类型约束

- 在使用template<>时，不使用typename/class，而使用类型约束
  template <convertable_to<bool> T>　void handle(const T& t);
  template <Incrementable T>　void process(const T& t);
- 使用整型参数调用process可以通过编译，但是用string调用则会给出“约束不满足”的编译错误
- 使用require，格式为template <typename T> requires *constant_expression*，其中*constant_expression*可以是
  template <typename T> requires Incrementable<T> //任意得到布尔类型的常量表达式，
  template <typename T> requires convertible_to<T,bool> //预定义的标准概念
  template <typename T> requires requires(T x) {x++; ++x; } //一个require表达式
  template <typename T> requires (sizeof(T)==4) //任何产生布尔值的常量表达式
  template <typename T> requires Incrementable<T> && Decrementable<T> //使用&&和||进行组合
  template <typename T> requires is_arithmetic_v<T> //类型萃取
- 还可以在函数头后指定require子句，即后置require子句
  template <typename T>
  void process(const T& t) requires Incrementable<T>;
- 对于任意模板，不可避免需要满足一些类型参数的约束，应对其进行类型约束，从而编译器可以进行检查

约束包含

- 可以通过不同的类型约束来重载函数模板，编译器将始终使用具有最具体约束的模板；更具体的约束将包含/暗示较小的约束

```
template <typename T>requires integral<T>
void process(const T& t) {cout<<“integral<T>”<<endl;}
template <typename T> requires (integral<T> && sizeof(T)==4)
void process(const T& t) {cout<<“integral<T> && sizeof(T)==4”<<endl;}
```

- 若所用系统int为32位，short16位，则

```
process(int {1}); //输出integral<T> && sizeof(T)==4
process(short {2});  // integral<T>
```

- 满足概念B的类型必然满足概念A，则概念A包含概念B
- 包含只在语法层面而不是语义层面，例如sizeof(T)>4在语义上比sizeof(T)>=4更具体，但在语法上前者不会包含后者

类模板的类型约束

- 新的GameBoard类模板，要求模板类型参数是GamePiece的派生类

```
template <std::derived_from<GamePiece> T>
//或者 template <typename T> requires std::derived_from<T, GamePiece>
class GameBoard : public Grid<T>  {
public: explicit GameBoard(size_t width = Grid<T>::DefaultWidth, size_t height = Grid<T>::DefaultHeight);
        void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest);
};

template <std::derived_from<GamePiece> T>
GameBoard<T>::GameBoard(size_t width, size_t height) : Grid<T>{ width, height } {    }

template <std::derived_from<GamePiece> T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest) {
        Grid<T>::at(xDest, yDest) = std::move(Grid<T>::at(xSrc, ySrc));
        Grid<T>::at(xSrc, ySrc).reset(); // Reset source cell
        // 或者 this->at(xDest, yDest) = std::move(this->at(xSrc, ySrc));
        // this->at(xSrc, ySrc).reset();
}
```

类方法的类型约束

- 可以对类模板的特定方法添加额外的约束。例如move方法进一步要求类型T是可移动的

```
template <std::derived_from<GamePiece> T>
class GameBoard : public Grid<T>  {
public:
        explicit GameBoard(size_t width = Grid<T>::DefaultWidth, size_t height = Grid<T>::DefaultHeight);
        void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest) requires std::movable<T>;
};
```

- 此时方法定义时也要加require

```
template <std::derived_from<GamePiece> T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest) requires std::movable<T>
{ … }
```

- 注：基于前面讨论的选择性实例化，仍然可以使用非移动类型的GameBoard，只要不使用其move方法

模板特化的类型约束

- 可以为类模板编写特化，可以重载函数模板使特定类型有特定实现，还可以一类满足特定条件的类型进行特化，以下是常规的Find函数模板

```
template <typename T>
size_t Find(const T& value, const T* arr, size_t size) {
        for (size_t i{ 0 }; i < size; i++)  if (arr[i] == value)  return i; // Found it; return the index.
        return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

- 该模板使用==进行比较，但对于浮点数，这种方式不太合适，为此进行特化

```
template <std::floating_point T>
size_t Find(const T& value, const T* arr, size_t size)  {
        for (size_t i{ 0 }; i < size; i++)  if (AreEqual(arr[i], value))  return i; // Found it; return the index.
        return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

- 辅助函数模板AreEqual，也要使用类型约束

```
template <std::floating_point T>
bool AreEqual(T x, T y, int precision = 2)  {
        return fabs(x - y) <= numeric_limits<T>::epsilon() * fabs(x + y) * precision
                        || fabs(x - y) < numeric_limits<T>::min(); // The result is subnormal.
}
```