IR自动生成
实验讲解

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年03月20日

CONTENT

- C++ 知识回顾
- LLVM 简介
- Light IR C++ 库
- IR 自动化生成框架

STL

STL (Standard Template Library 标准模板库)，包含了许多常用的数据结构

STL	实现
std::vector<T>	可变长数组
std::map<T, T>	由红黑树实现的有序关联容器
std::set<T>	由红黑树实现的有序集合
std::unordered_map<T, T>	由哈希表实现的无序关联容器
std::list<T>	链表

这里的 std 是 C++ 中的命名空间，可以防止标识符的重复

同时，这些容器都是模板，需要**实例化**

STL vector

- vector是STL容器中的一种常用的容器，和数组类似，由于其大小(size)可变，常用于数组大小不可知的情况下来替代数组
- vector是为了实现动态数组而产生的容器
- vector也是一种顺序容器，在内存中连续排列，因此可以通过下标快速访问，时间复杂度为O(1)
- 连续排列也意味着大小固定，数据超过vector的预定值时vector将自动扩容

STL vector

- 使用vector需要包含头文件

```
#include <vector>
```

STL vector

- Vector本质是类模板，可以存储任何类型数据。在数据声明前需要加上数据类型

- 例如，声明一个int型vector数组

```
vector<int> arr1;                                    //一个空数组
vector<int> arr2 {1, 2, 3, 4, 5}; //包含1、2、3、4、5五个变量
vector<int> arr3(4);                          //开辟4个空间，值默认为0
vector<int> arr4(5, 3);                       //5个值为3的数组
vector<int> arr5(arr4);                       //将arr4的所有值复制进去，和arr4一样
vector<int> arr6(arr4.begin(), arr4.end());       //将arr4的值从头开始到尾复制
vector<int> arr7(arr4.rbegin(), arr4.rend());     //将arr4的值从尾到头复制
```

STL vector

- 遍历vector
  - 1 迭代器访问

```
vector<int> arr2 {1, 2, 3, 4, 5};

//迭代器：vector<int>::iterator
for (vector<int>::iterator it = arr.begin(); it != arr.end(); it++)
{
    cout << *it << endl;
}
//迭代器：vector<int>::reverse_iterator
for (vector<int>::reverse_iterator it = arr.rbegin(); it != arr.rend(); it++)
{
    cout << *it << endl;
}
```

STL vector

- 遍历vector
  - 2 下标访问

```
vector<int> arr2 {1, 2, 3, 4, 5};

for (int i = 0; i < arr.size(); i++)
{
        cout << arr[i] << endl;
}
```

STL vector

- 遍历vector
  - 3 for循环

```
vector<int> arr2 {1, 2, 3, 4, 5};

for (auto num : arr)
{
        cout << num << endl;
}
```

STL vector

- vector容量和大小
  - size表示当前有多少个元素
  - capacity是可容纳的大小
  - resize改变size的大小，reserve改变capacity的大小
  - shrink_to_fit减小capacity到size

```
vector<int> arr;
arr.resize(4);
arr.reserve(6);
cout << arr.size() << " " << arr.capacity() << endl;
cout << "##############################" << endl;
arr.shrink_to_fit();
cout << arr.size() << " " << arr.capacity() << endl;
```

STL vector

- vector常用方法
  - push_back、pop_back

```
vector<int> arr;
for (int i = 0; i < 5; i++)
{
    arr.push_back(i);
}
for (int i = 0; i < 5; i++)
{
    arr.pop_back();
}
```

STL vector

- vector常用方法
  - insert

```
//在指定位置插入值为val的元素。在arr的头部插入值为10的元素
vector<int> arr;
arr.insert(arr.begin(), 10);

//在指定位置插入n个值为val的元素。从arr的头部开始，连续插入3个值为10元素
vector<int> arr;
arr.insert(arr.begin(), 3, 10);

//在指定位置插入区间[start, end]的所有元素
//从arr的头部开始，连续插入arrs区间[begin, end]的所有元素
vector<int> arr;
vector<int> arrs = { 1, 2, 3, 4, 5 };
arr.insert(arr.begin(), arrs.begin(), arrs.end());
```

STL vector

- vector常用方法
  - erase
    - erase通过迭代器删除某个或某个范围的元素，并**返回下一个元素的迭代器**

```
vector<int> arr{1, 2, 3, 4, 5};

//删除arr开头往后偏移两个位置的元素，即arr的第三个元素，3
arr.erase(arr.begin() + 2);

//删除arr.begin()到arr.begin()+2之间的元素，删除两个
//即删除arr.begin()而不到arr.begin()+2的元素
arr.erase(arr.begin(), arr.begin() + 2);
```

STL vector

- vector常用方法
  - clear
    - 清空整个vector，size变为0，但空间仍然存在

```
vector<int> arr{1, 2, 3, 4, 5};

arr.clear();
```

STL vector

- vector常用方法
  - 二维操作
    - 嵌套定义vector

```
//定义一个空的二维vector
vector<vector<int>> arr;

//定义一个5行3列值全为1的二维vector
vector<vector<int>> arr(5, vector<int>(3, 1));
```

STL vector

- vector常用方法
  - 二维操作
    - 访问
      - 和二维数组一样通过 [] [] 访问即可

```
for (int i = 0; i < arr.size(); i++)
{
        for (int j = 0; j < arr[0].size(); j++)
        {
                cout << arr[i][j] << endl;
        }
}
```

STL vector

- vector常用方法
  - 二维操作
    - 访问
      - 和二维数组一样通过 [] [] 访问即可
      - 采用范围for

```
for (auto nums : arr)
{
    for (auto num : nums)
    {
        cout << num << endl;
    }
}
```

STL vector

- vector扩容原理

```
vector<int> arr;
for (int i = 0; i < 20; i++)
{
    arr.push_back(i);
    cout << arr.size() << " " << arr.capacity() << endl;
}
```

STL vector

- vector扩容原理
  - 运行结果

```
ubuntu@VM7939-test:~/Desktop$ ./a.out
1  1
2  2
3  4
4  4
5  8
6  8
7  8
8  8
9  16
10  16
11  16
12  16
13  16
14  16
15  16
16  16
17  32
18  32
19  32
20  32
```

STL的实例化

- 例如 AST.hpp 中：

```
struct ASTProgram : ASTNode {
    virtual Value* accept(ASTVisitor &) override final;
    virtual ~ASTProgram() = default;

    std::vector<std::shared_ptr<ASTDeclaration>> declarations;
};
```

这里声明了一个可变长数组(vector)，其中的每一个元素类型为指向 ASTDeclaration 类的共享指针

STL

STL (Standard Template Library 标准模板库)，包含了许多常用的数据结构

STL	实现
std::vector<T>	可变长数组
std::map<T, T>	由红黑树实现的有序关联容器
std::set<T>	由红黑树实现的有序集合
std::unordered_map<T, T>	由哈希表实现的无序关联容器
std::list<T>	链表

编程中经常使用到key/value的形式表示数据之间的关系
STL库提供了map和unordered_map容器，对应处理key/value数据的存储、查找等问题
map内key/value是有序的，unordered_map则是无序的
使用时需要引入<map>头文件

STL 使用举例


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

int main() {
...
```

STL 使用举例

```
map<int, string> node;                          // 定义变量
map<int, string>::iterator iter;                        // 定义迭代器 iter
node[16] = "张三";                                      // 以数组下标的形式
node.insert(pair<int, string>(28, "李四"));  // 直接插入键值对，pair定义了一个键值对，
                                                        // 对应map的key和value。

node[78] = "陆七";
node[58] = "陈二";
node[39] = "王五";

for(iter = node.begin(); iter != node.end(); ++iter) {
        cout<<"工号"<<iter->first<<": "<<iter->second<<", "<<endl;
}
// 输出“工号16：张三，工号28：李四，39：王五，工号58：陈二，工号78：陆七”
```

STL 使用举例 (cont.)

```
node.erase(58);                                    // 使用key删除key=58的节点

iter = node.find(78);                              // 使用迭代器查找key=78的节点
node.erase(iter);                                  // 删除key=78的节点

node[28] = "赵四";                                 // 仅能修改 value 的值

for(iter = node.begin(); iter != node.end(); ++iter) {
        cout<<"工号"<<iter->first<<"："<<iter->second<<"，"<<endl;
}
// 输出“工号16：张三，工号28：赵四，39：王五”
```

String

C++ 提供了更易用的 **std::string** 以处理字符串，可以支持通过 + 拼接，还提供了许多方法：

- length: 返回字符串长度
- push_back(c): 在字符串末尾添加一个字符 c
- append(str): 在字符串末尾添加字符串 str
- substr: 取子串

String 常用成员方法与运算符

```
std::string str = "Hello";
std::cout << str;                          // Hello
std::cout << str.size();                   // 5 (length和size完全相同)
std::cout << str[0];                             // H
std::cout << str.append(" World");  // Hello World
std::cout << str + "!";                    // Hello World!
std::cout << str.substr(6, 5);             // World
str.push_back( ‘!’ );
std::cout << str;                          // Hello World!
```

Class

Class是 C++ 面向对象的基础，相当于对 C 中的结构体的扩展

除了保留了原来的结构体成员（即成员对象），增加了成员函数、访问控制、继承和多态等

Class

```
class AST {
  public:
    AST() = delete;
    AST(syntax_tree *);                                          <-成员函数
    ...
  private:
    ASTNode *transform_node_iter(syntax_tree_node *); <-成员变量
    std::shared_ptr<ASTProgram> root = nullptr;                  <-成员变量
};
```

Class

Class是 C++ 面向对象的基础，相当于对 C 中的结构体的扩展

除了保留了原来的结构体成员（即成员对象），增加了成员函数、访问控制、继承和多态等

访问控制：用 public/private 标签指定成员为公开/私有

私有成员只有该类的成员函数能访问

对使用者隐藏实现的细节，只提供想要公开的接口

Class

```
class AST {
  public:                          // 以下成员是公有的
    AST() = delete;
    AST(syntax_tree *);
    ...
  private:                         // 以下成员是私有的
    ASTNode *transform_node_iter(syntax_tree_node *);
    std::shared_ptr<ASTProgram> root = nullptr;
};
```

继承

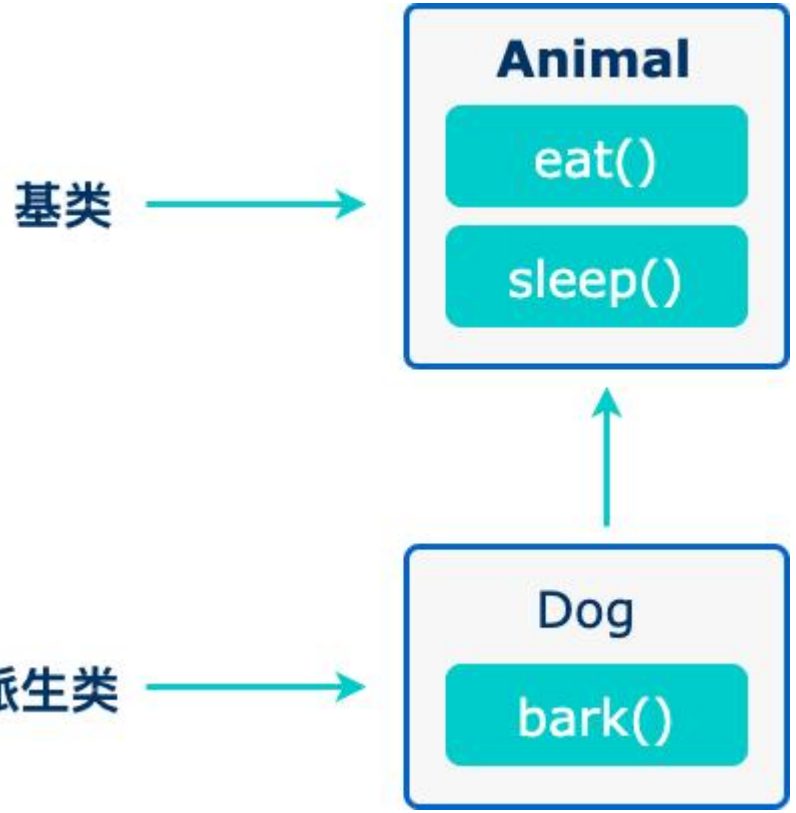
类的继承是一种面向对象语言常用的代码复用方法，也是一种非常直观的抽象方式

```
struct ASTNode {
    virtual Value* accept(ASTVisitor &) = 0;                    //代表没有实现
    virtual ~ASTNode() = default;
};

struct ASTProgram : ASTNode {
    virtual Value* accept(ASTVisitor &) override final;         //重写父类成员函数
    virtual ~ASTProgram() = default;                            //使用默认析构函数
    std::vector<std::shared_ptr<ASTDeclaration>> declarations;  //新的成员
};
```

继承

- 面向对象程序设计中最重要的一个概念
- 继承允许我们依据另一个类来定义一个类，使得创建和维护一个应用程序变得更容易
- 达到了重用代码功能和提高执行效率的效果



- 被继承的类称为父类或基类
- 继承的类称为子类或派生类
- 一个类可以派生自多个类

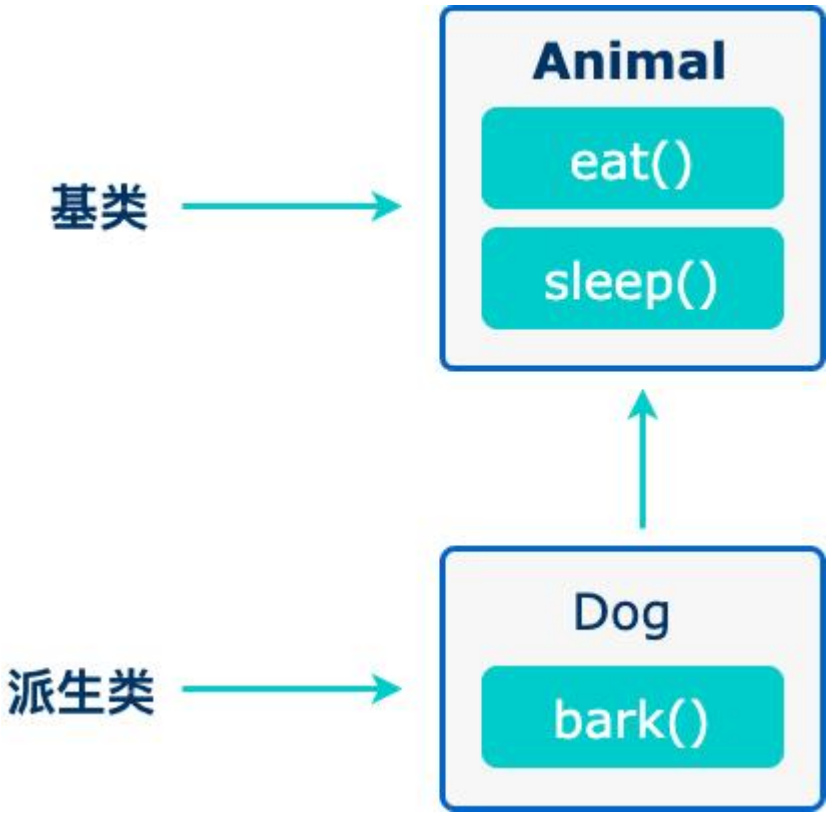
继承

```
// 基类
class Animal {
    eat() 函数
    sleep() 函数};

//派生类
class Dog : public Animal {
    bark() 函数
};


...

Dog D1;
D1.eat();
```



继承

```
#include <iostream>
 using namespace std;
 // 基类
class Shape {
   public:
      void setWidth(int w)       {   width = w; }
      void setHeight(int h)      {   height = h; }
   protected:      int width;      int height;};

// 派生类
class Rectangle: public Shape{
   public:
      int getArea() {return (width * height);}
};
```

```
int main(void){
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
    // 输出对象的面积
   cout << "Total area: " << Rect.getArea() <<
endl;
    return 0;
}
```

继承

```
#include <iostream>
 using namespace std;
 // 基类
class Shape {
   public:
      void setWidth(int w)       {    width = w;}
      void setHeight(int h)      {    height = h;      }
   protected:      int width;       int height;};

// 派生类
class Rectangle: public Shape{
   public:
      int getArea() {return (width * height);}
};
```

```
int main(void){
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
    // 输出对象的面积
   cout << "Total area: " << Rect.getArea() << endl;
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total area: 35
```

虚函数和多态

多态

不同继承关系的类对象，去调用同一函数，产生了不同的行为

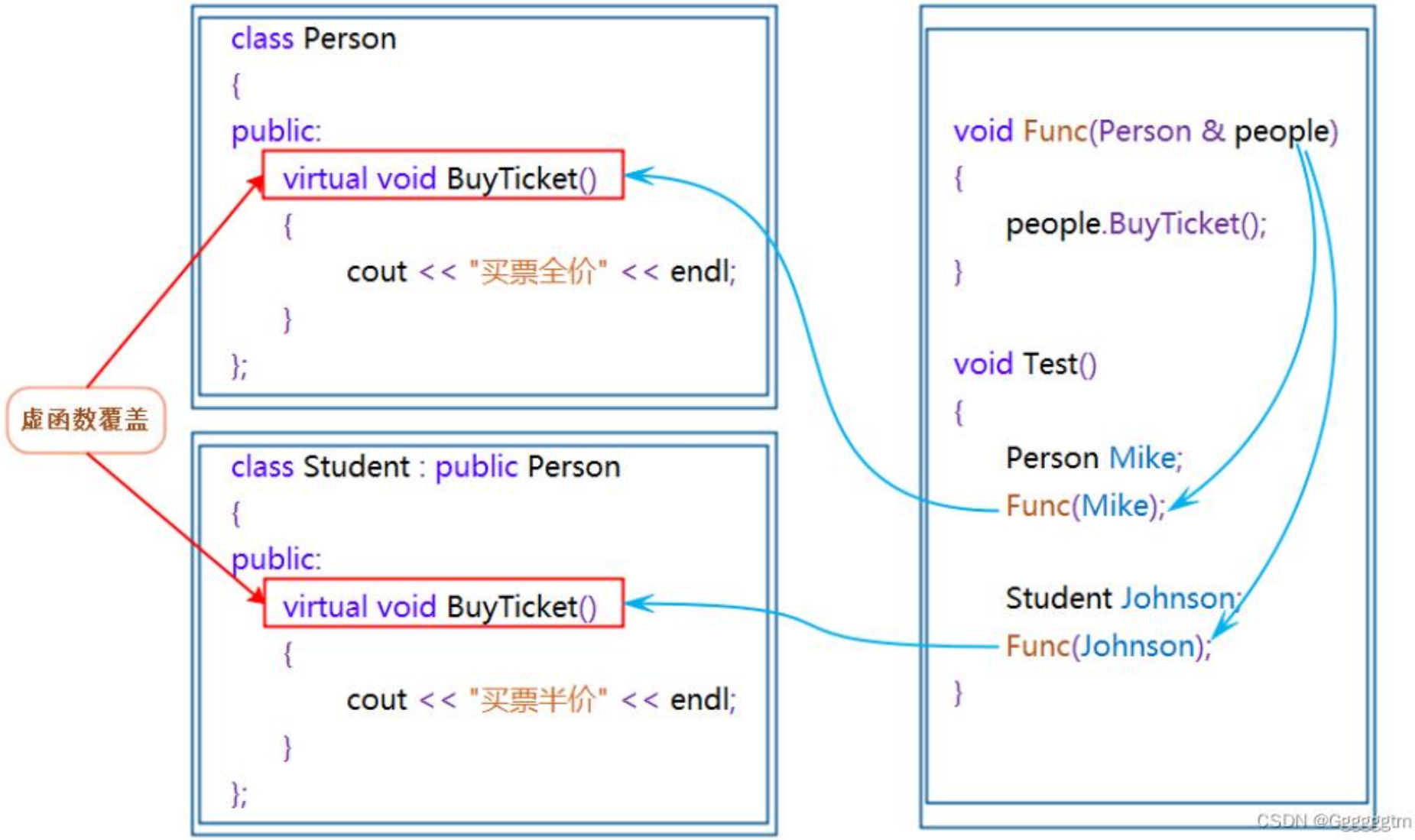
继承中要构成多态还有两个条件：

必须通过基类的指针或者引用调用虚函数

被调用函数必须是虚函数，且派生类必须对基类虚函数进行重写

虚函数和多态

多态：同一个方法（接口）调用，由于对象不同可能会有不同的行为。



虚函数和多态

- 多态的条件中提到了虚函数
- 虚函数，就是被virtual修饰的类成员函数。

```
class Person {
public:
        virtual void BuyTicket()
        { cout << "买票-全价" << endl; }
};
```

上述的代码中，成员函数 BuyTicket() 即为虚函数。

虚函数和多态

子类的指针可以给基类指针赋值

在基类指针上调用虚函数时，会通过虚函数表查找到对应的函数实现

```
// in ast.hpp
struct ASTNode { virtual Value* accept(ASTVisitor &) = 0; };
struct ASTProgram : ASTNode { virtual Value* accept(ASTVisitor &) override final; };
struct ASTParam : ASTNode { virtual Value* accept(ASTVisitor &) override final; };
// in ast.cpp
Value* ASTProgram::accept(ASTVisitor &visitor) { return visitor.visit(*this); }
Value* ASTParam::accept(ASTVisitor &visitor) { return visitor.visit(*this); }
// in cminusf_builder.hpp
class CminusfBuilder : public ASTVisitor{
virtual Value *visit(ASTProgram &) override final;
virtual Value *visit(ASTParam &) override final; }
// in cminusf_builder.cpp
Value* CminusfBuilder::visit(ASTProgram &node) { /* some codes  */ }
Value* CminusfBuilder::visit(ASTProgram &node) { /* other codes  */ }
```

类型转换

既然子类的指针可以赋值给基类，为了得到基类指针的实际类型，需要进行类型转换

在C++中，提供两种编程机制获取对象的实际类型

static_cast和dynamic_cast

static_cast

- **static_cast是一个强制类型转换操作符，用于**
  - 不同变量类型之间的转换，例如short转int、int转double等
  - void指针和具体类型指针之间的转换，例如void *转int *、char *转void *等
  - 有转换构造函数或者类型转换函数的类与其它类型之间的转换

- **static_cast不能用于无关类型之间的转换**
  - int *转double *、Inst *转int *等
  - int和指针之间的转换

类型转换举例

```
int main(){
        int m = 0xffd7;
        long n = static_cast<long>(m);                          // 宽转换，没有信息丢失
        char ch = static_cast<char>(m);                         // 窄转换，可能会丢失信息
        int *p1 = static_cast<int*>(malloc(10 * sizeof(int)));          // 将void指针转换为具体类型指针
        void *p2 = static_cast<void*>(p1);                              // 将具体类型指针转换为void指针
        // 下面的用法是错误的
        // float *p3 = static_cast<float*>(p1);                         // 不能在两个具体类型的指针之间转换
        // float * p34 = static_cast<float*>(100);              // 不能将整数转换为指针类型
        cout << m << endl;                                              // 65335
        cout << ch << endl;                                             // 7
        cout << p1 << endl;                                             // 0x5614372972b0
        cout << p2 << endl;                                             // 0x5614372972b0
        return 0;}
```

dynamic_cast

- 和static_cast一样，dynamic_cast用于类型的转换

- dynamic_cast可以用于基类和派生类指针之间的相互转换
  - 将派生类对象赋值给赋值给基类引用称为向上转型
  - 反之称为向下转型
  - 并且能够检查类型是否符合转换

- 语法形式

  dynamic_cast < new-type > (expression)

new-type和expression必须同时是指针类型或者引用类型
即dynamic_cast只能转换指针类型和引用类型。

dynamic_cast

- 对于指针类型，如果转换失败将返回NULL
- 对于引用，如果转换失败将抛出std::bad_cast异常

- 向上转型时，只要待转换的两个类型之间存在继承关系，并且基类包含了虚函数就一定能转换成功

- 示例

```
Derived *down = new Derived();
Base *up = dynamic_cast<Base*>(down);
```

派生类指针down被向上转型为基类指针并赋值给pb

dynamic_cast

- 向下转型是有风险的
- 安全的向下转型
  - 声明为基类的指针实际指向的是派生类对象，这时就可以将该指针向下转型为派生类指针

- 示例

```
Base *up= new Derived();
Derived *down = dynamic_cast<Derived*>(up);
```

基类指针up被向下转型为派生类指针并赋值给pd，安全的原因是up实际指向的是派生类对象

类型转换举例

```
class B {
public:
    virtual void fun() {
        cout << "base fun" << endl;
    }
};
class D: public B {
public:
    void fun() {
         cout << "derived fun" << endl;
    }
};
```

类型转换举例

```
class B {
public:
    virtual void fun() {
        cout << "base fun" << endl;
    }
};
class D: public B {
public:
    void fun() {
         cout << "derived fun" << endl;
    }
};
```

```
// In Main
B b;
D d;
D *ptr = new D;
B *p = new D;
b.fun();
d.fun();
ptr->fun();
p->fun();
```

```
base fun
derived fun
derived fun
derived fun
```

类型转换举例

```
// In Main
...
B* ptrb = dynamic_cast<B*>(ptr);          // upcasting向上转类型
cout << "ptr is " << ptr << endl;
cout << "ptrb is " << ptrb << endl;
D* pd = dynamic_cast<D*>(p);              // downcasting向下转类型
cout << "p is " << p << endl;
cout << "pd is " << pd << endl;
B* pb = dynamic_cast<B*>(p);
cout << "pb is " << pb << endl;
```

```
ptr is 0x632e70
ptrb is 0x632e70
p is 0x632e90
pd is 0x632e90
pb is 0x632e90
```

重载

- **C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载**

- **重载**
  - 一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同

函数重载示例

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

class printData
{
public:
        void print(int i) { cout << "整数为: " << i << endl; }
        void print(double f) { cout << "浮点数为: " << f << endl; }
        void print(char c[]) { cout << "字符串为: " << c << endl; }
};
```

函数重载示例

```
int main(void) {
        printData pd;
        pd.print(5);                    // 输出整数
        pd.print(500.263);              // 输出浮点数
        char c[] = "Hello C++";
        pd.print(c);                    // 输出字符串
        return 0;
}
```

智能指针

早期的 C++ 堆上分配的内存空间需要手动释放，否则会导致内存泄漏

```
int* p = new int(50);
// ...
delete p;
```

智能指针

- C++设计了智能指针
  - 将回收的程序放在智能指针的析构函数中
  - 利用智能指针是创建在栈空间上的对象，它会在生命周期结束的时候由系统自动调用析构函数并回收，这样就做到了自动delete释放内存

```
{
        std::unique_ptr<int> sptr = std::make_unique<int>(50);
        //...
        // 离开sptr的作用域的时候，程序会自动释放智能指针申请的内存
 }
```

智能指针

- **并不是所有指针都改为智能指针，很多时候原始指针要方便**

- **智能指针分类**
  - 独占指针unique_ptr
  - 共享指针shared_ptr
  - 弱指针weak_ptr
  - auto

智能指针

- 独占指针unique_ptr
  - 任何时刻，只能有一个指针管理内存
- 当指针超出作用域时，内存将自动释放
- 该类型指针不可copy，只能move

智能指针

- 常见的独占指针unique_ptr构造

```
// unique_ptr<T> up; 空的unique_ptr，可以指向类型为T的对象unique_ptr<Test> t1;

// unique_ptr<T> up1(new T());          定义unique_ptr,同时指向类型为T的对象
unique_ptr<Test> t2(new Test);

// unique_ptr<T[]> up; 空的unique_ptr，可以指向类型为T[的数组对象
unique_ptr<int[]> t3;

// unique_ptr<T[]> up1(new T[]); 定义unique_ptr,同时指向类型为T的数组对象unique_ptr<int[]> t4(new int[5]);
```

智能指针

- 常见的独占指针unique_ptr赋值

```
unique_ptr<Test> t7(new Test);
unique_ptr<Test> t8(new Test);
t7 = std::move(t8); // 必须使用移动语义，结果，t7的内存释放，t8的内存交给t7管理
t7->doSomething();
```

智能指针

- 常见的独占指针unique_ptr主动释放对象

```
unique_ptr<Test> t9(new Test);

t9 = NULL;
t9 = nullptr;
t9.reset();
```

智能指针

- **常见的独占指针unique_ptr放弃对象的控制权**

```
t9.release();
//t9放弃对指针的控制权，返回指针，并将t9置空
```

- **常见的独占指针unique_ptr重置**

```
t9.reset(new Test);    //释放指向的对象
```

智能指针示例

```
unique_ptr<string> p1(new string("I'm Li Ming!"));
unique_ptr<string> p2(new string("I'm age 22."));

cout << "p1：  " << p1.get() << endl;                    // 00E183A8
cout << "p2：  " << p2.get() << endl;                    // 00E18318
p1 = p2;                                                 // 禁止左值赋值
unique_ptr<string> p3(p2);                               // 禁止左值赋值构造

unique_ptr<string> p3(std::move(p1));
p1 = std::move(p2);                                      // 使用move赋值
cout << "p1 = p2 赋值后：  " << endl;
cout << "p1：  " << p1.get() << endl;                    // 00E18318
cout << "p2：  " << p2.get() << endl;                    // 00000000
```

智能指针

- 共享指针shared_ptr

- 每个shared_ptr对象关联有一个共享的引用计数
  - 当复制一个shared_ptr，将其引用计数值加1；
  - shared_ptr提供unique()和use_count()两个函数来检查其共享的引用计数值，前者测试该shared_ptr是否是唯一拥有者（即引用计数值为1），后者返回引用计数值；
  - 当shared_ptr共享的引用计数降低到0的时候，所管理的对象自动被析构（调用其析构函数释放对象）。

智能指针示例2

```
class Test{
public: ...
        Test(string s)          { str = s; cout<<"Test creat\n";}
        ~Test()                 {cout<<"Test delete:"<<str<<endl;}
        string& getStr()        { return str; }
        void setStr(string s)   { str = s; }
        void print()            { cout<<str<<endl; }
private:
        string str;
};

unique_ptr<Test> fun()          { return unique_ptr<Test>(new Test("789")); }
```

智能指针示例2

```
shared_ptr<Test> ptest(new Test("123"));          // 调用构造函数输出Test create
shared_ptr<Test> ptest2(new Test("456"));         // 调用构造函数输出Test creat
cout<<ptest2->getStr()<<endl;                     // 输出456
cout<<ptest2.use_count()<<endl;                   // 显示此时资源被几个指针共享，输出1
ptest = ptest2;                  // "456"引用次数加1，“123”销毁，输出Test delete：123
ptest->print();                                   // 输出456
cout<<ptest2.use_count()<<endl;                   // 该指针指向的资源被几个指针共享，输出2
cout<<ptest.use_count()<<endl;                    // 输出2
ptest.reset();                                    // 重新绑定对象，绑定一个空对象，当时此时指针指向的对象还有其他指针能指向就不会释放该对象的内存空间
ptest2.reset();                                   // 此时“456”销毁，此时指针指向的内存空间上的指针为0，就释放了该内存，输出Test delete
cout<<"done !\n";
```

Auto

auto 关键字可以用于当类型已知时自动推断类型，
常用于声明迭代器遍历容器

```
std::vector<std::string> v;
v.push_back("compile");
auto s = v.front();
```

这里 s 就是 std::string 类型

Auto 常用技术

for (auto ...) 语法是 C++11 引入的范围基于 for 循环，用于简化遍历容器的代码

```
for (auto element : container) {
    // 使用 element
}
```

- auto：自动推断循环变量的类型
- element：循环变量，用于访问容器中的每个元素
- container：要遍历的容器，如数组、std::vector、std::list 等

一起努力
打造国产基础软硬件体系！

徐　伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年03月20日