



Lab2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

2024年10月12日

CONTENT



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



C++ 知识回顾

编译原理课程组
中国科学技术大学

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

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

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

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

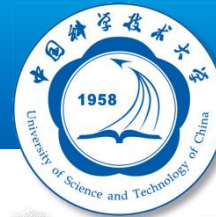


```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {

...
}
```

STL 使用举例



```
map<int, string> node;
map<int, string>::iterator iter;
node[16] = "张三";
node.insert(pair<int, string>(28, "李四"));

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”
```

// 定义变量
// 定义迭代器 iter
// 以数组下标的形式
// 直接插入键值对, pair定义了一个键值对,
// 对应map的key和value。

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"
```

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 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

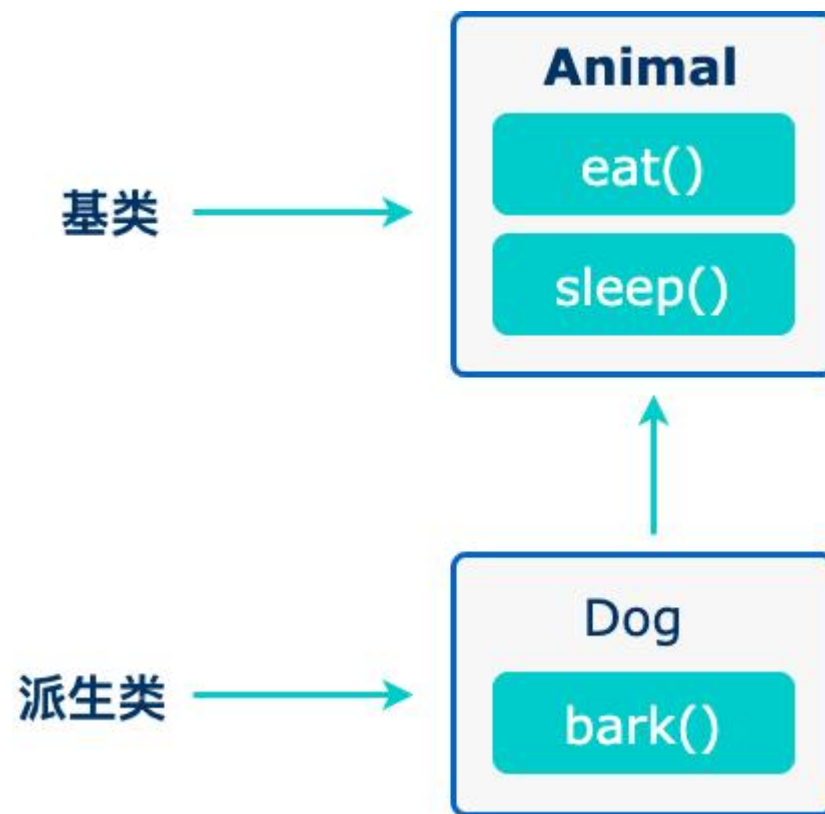
访问控制：用 public/private 标签指定成员为公开/私有
私有成员只有该类的成员函数能访问
对使用者隐藏实现的细节，只提供想要公开的接口

```
class 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;  
};
```

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  
};
```

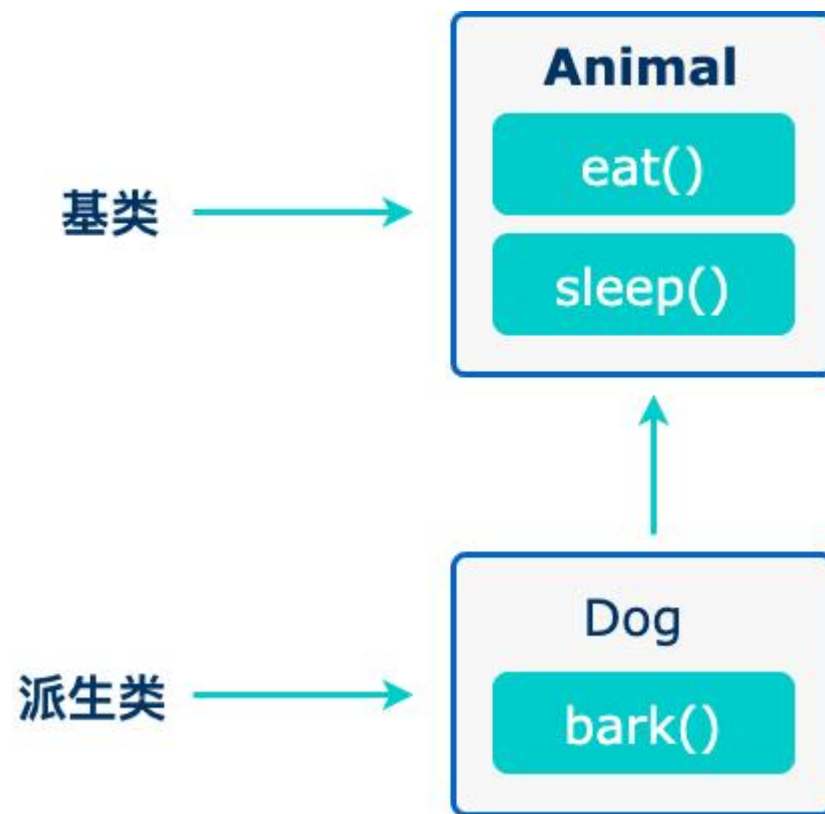

- 面向对象程序设计中最重要的一個概念
- 继承允许我们依据另一个类来定义一个类，使得创建和维护一个应用程序变得更容易
- 达到了重用代码功能和提高执行效率的效果
- 被继承的类称为父类或基类
- 继承的类称为子类或派生类
- 一个类可以派生自多个类



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

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

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



```
#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;
}
```

```
#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

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  
};
```

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：

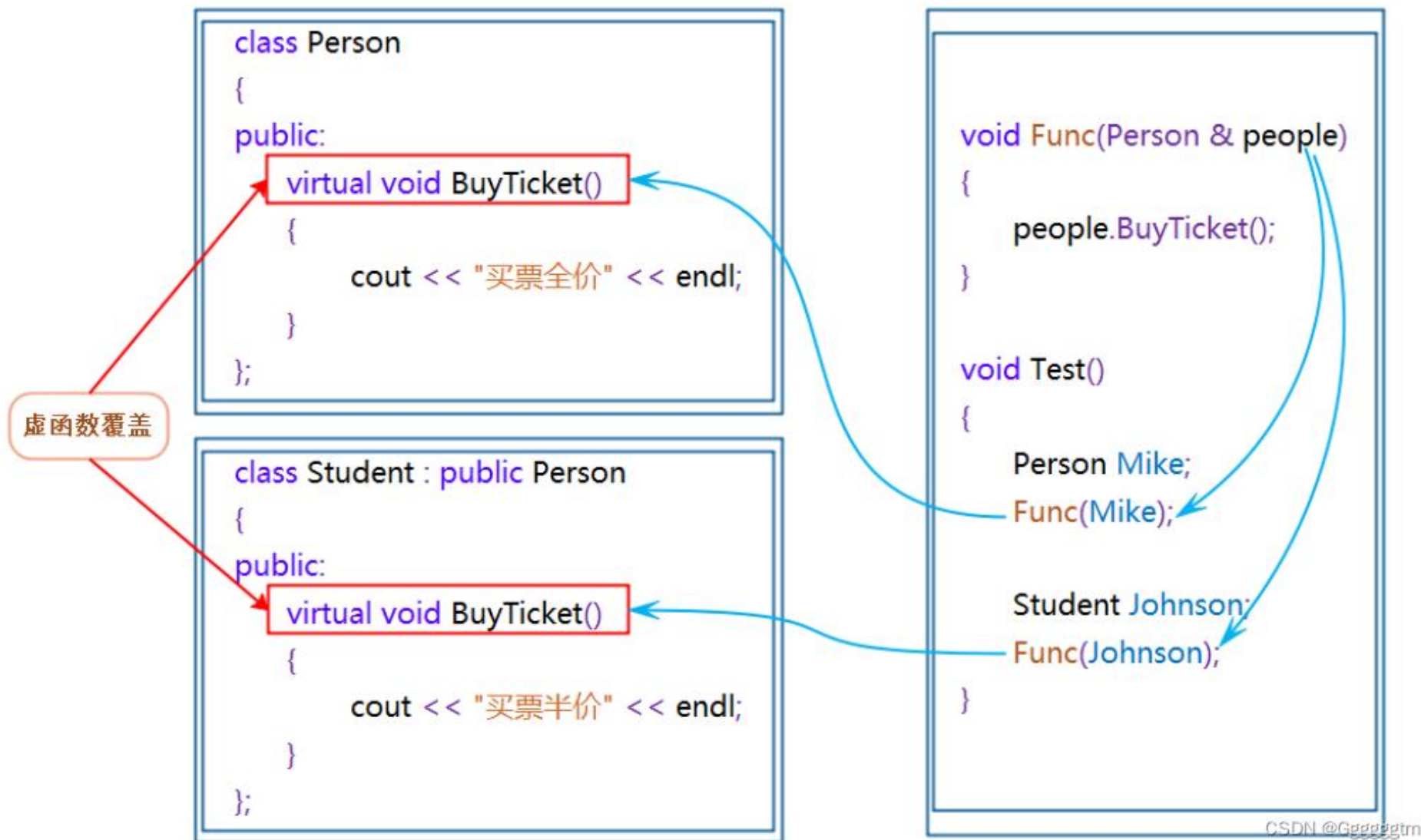
必须通过基类的指针或者引用调用虚函数

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。



- 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。

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(ASTParam &node) { /* other codes */ }
```

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**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 = 0xff07;
    long n = static_cast<long>(m);
    char ch = static_cast<char>(m);
    int *p1 = static_cast<int*>(malloc(10 * sizeof(int)));
    void *p2 = static_cast<void*>(p1);
    // 下面的用法是错误的
    // float *p3 = static_cast<float*>(p1);
    // float * p34 = static_cast<float*>(100);
    cout << m << endl;
    cout << ch << endl;
    cout << p1 << endl;
    cout << p2 << endl;
    return 0;}

// 宽转换, 没有信息丢失
// 窄转换, 可能会丢失信息
// 将void指针转换为具体类型指针
// 将具体类型指针转换为void指针

// 不能在两个具体类型的指针之间转换
// 不能将整数转换为指针类型
// 65287
// 7
// 0x5614372972b0
// 0x5614372972b0
```

- 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。**

- 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

- **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);  
cout << "ptr is " << ptr << endl;  
cout << "ptrb is " << ptrb << endl;  
D* pd = dynamic_cast<D*>(p);  
cout << "p is " << p << endl;  
cout << "pd is " << pd << endl;  
B* pb = dynamic_cast<B*>(p);  
cout << "pb is " << pb << endl;
```

// upcasting向上转类型

// downcasting向下转类型

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

- **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;  
}
```

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;
```

• 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  
}
```

- 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

- **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**

• 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]);

• 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();
```

- 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();
```

- 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
```

```
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
```

- **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 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

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



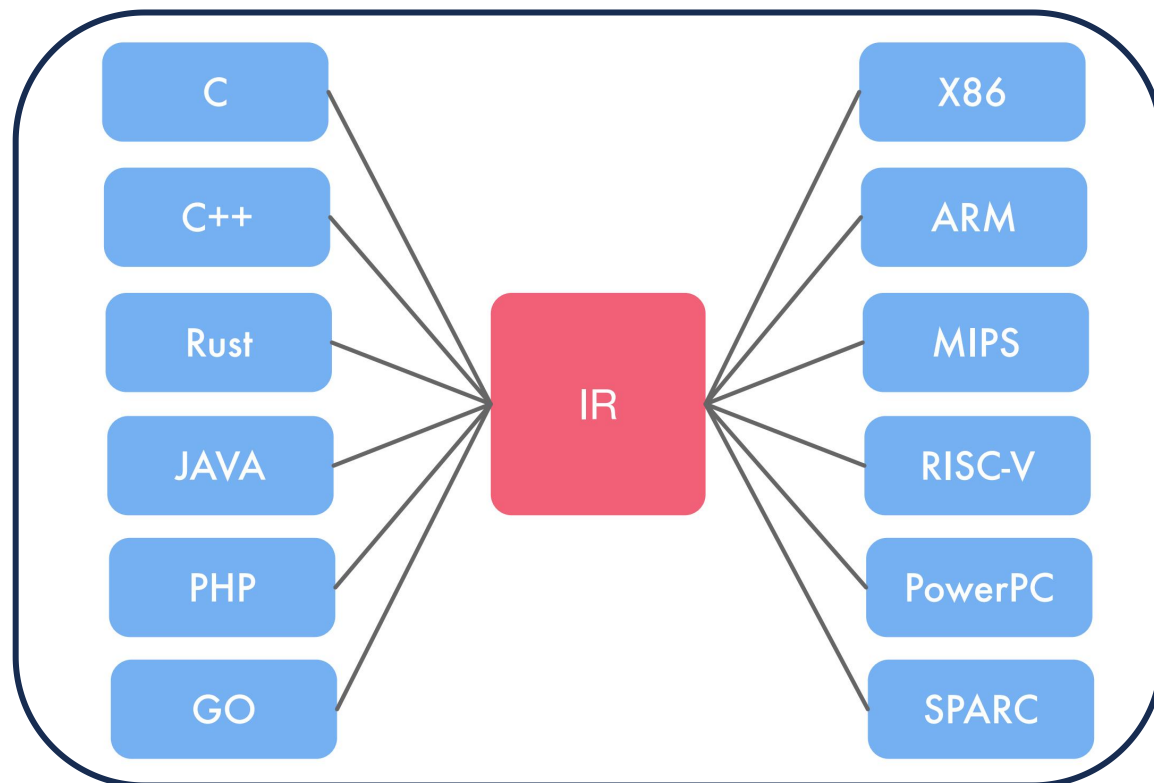
LLVM 简介

编译原理课程组
中国科学技术大学

传统编译器 - Compiler Collection



• 传统编译器 (GCC) 三段式架构



Compiler Collection



传统编译器面临的问题

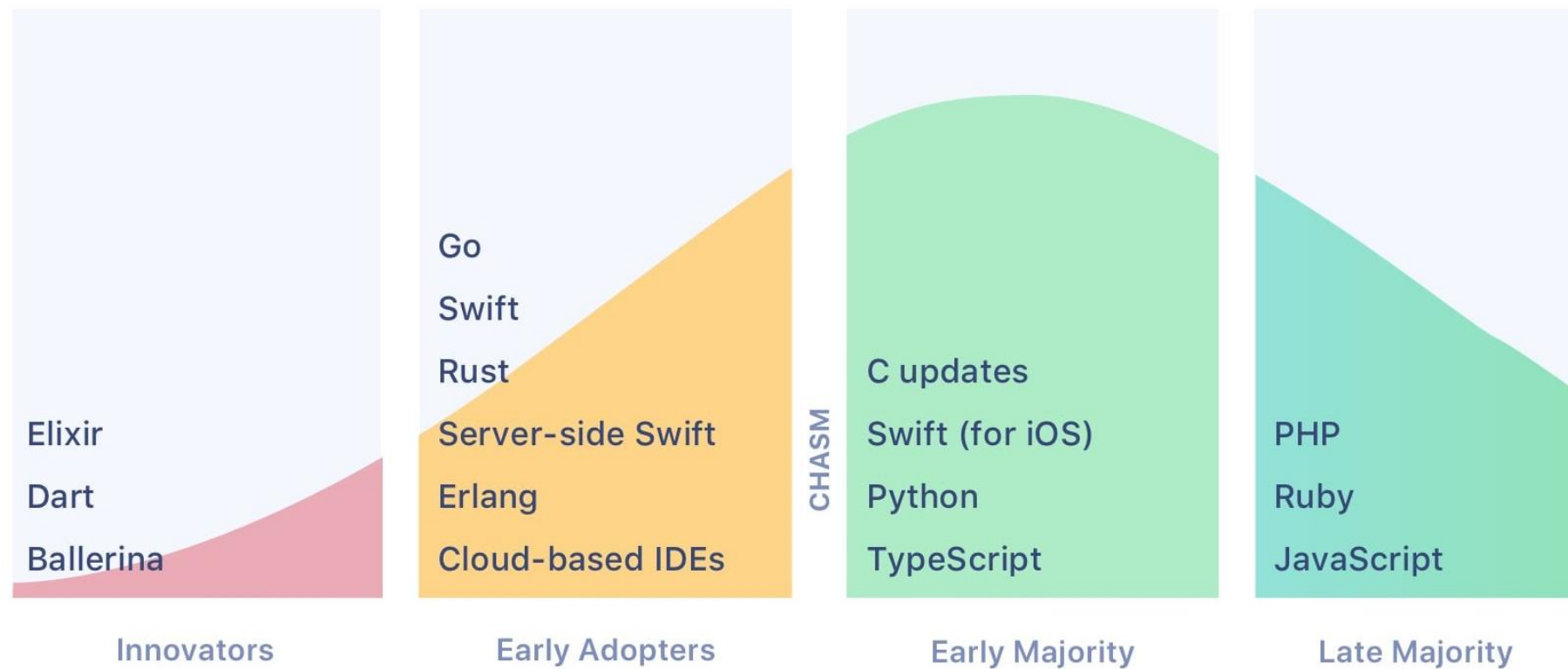


• 推陈出新的语言

Software Development
Programming Languages Trends 2019 Q3 Graph

<http://info.link/proglang2019>

InfoQ



• 推陈出新的语言、不断涌现新的目标平台

分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	MDMX , MIPS-3D	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

- 推陈出新的语言、不断涌现新的目标平台

分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	MDMX , MIPS-3D	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

- 问题：传统编译器饱受**分层**和**抽象漏洞困扰**，添加新语言与目标平台的支持时**源代码重用**的难度较大

- LLVM 制定了LLVM IR, 将编译器需要的功能**包装成库**, 解决了编译器**代码重用**的问题



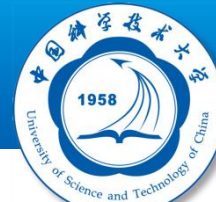
Compiler Collection



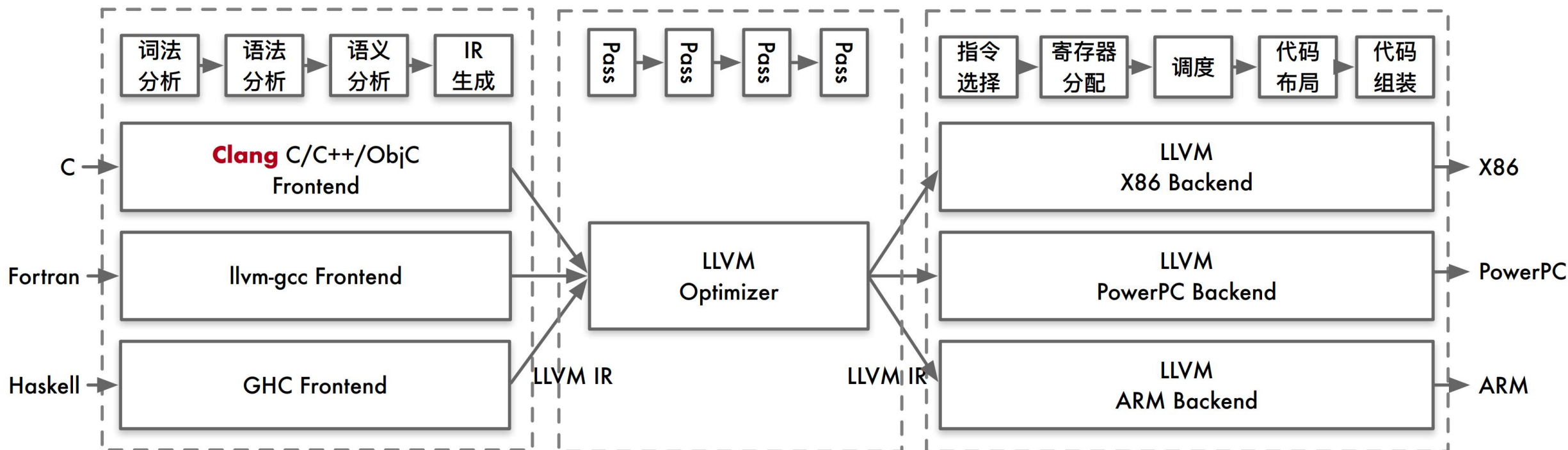
Libraries Collection

- LLVM 是一个编译器
- LLVM 是一个编译框架
- LLVM 是一系列编译工具
- LLVM 是一个编译工具链
- LLVM 是一个开源C++的实现
- LLVM 项目发展为一个巨大的编译器相关的工具集合

LLVM 架构



- LLVM 将很多编译器需要的功能**包装成库**，供其他编译器实现者根据需要使用或者扩展

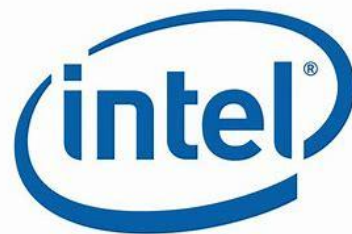


LLVM 在工业界的广泛使用



- 工业界使用者 (包括 Apple, Huawei, Intel, NVIDIA, Adobe, Sony...
30+ 公司使用 LLVM 开发相应产品)

Company	Description
Apple	iOS, macOS, tvOS and watchOS
Huawei	BiSheng Compiler for Huawei's Kunpeng servers
Intel	OpenCL* compiler, debugger
NVIDIA	OpenCL runtime compiler (Clang + LLVM)



LLVM 在学术界的广泛使用



- 学术界使用者（包括 **CMU, ETH, UIUC, UCLA, Stanford** ... 在内 **20+** 顶尖大学研究组使用 **LLVM** 完成相关研究项目）

Organization	Faculty	Description
CMU	David Koes	Principled Compilation
Stanford	Dawson Engler's Research Group	KLEE Symbolic Virtual Machine
ETH Zurich	Thomas Lenherr	Language-independent library for alias analysis
UCLA	Jason Cong	xPilot behavioral synthesis system

- 教育界使用者（包括 **CMU, ETH, UIUC, UCLA** ... 在内 **10+** 已公开的顶尖大学编译课程中使用 **LLVM** 工具链设计实验）

LLVM 在就业中的应用



• 很多互联网公司急需懂得 LLVM 实践人才

Program Language Engineer - LLVM Direction

Byte Dance 4.1

上海市 [+2地区](#)

languages... LLVM ecosystem... techniques, experience in LLVM, GCC, Go related compiler...

19 天前发布 · 更多.....

平头哥-编译器技术专家-AI软件-上海

阿里巴巴集团

上海市

对机器学习算法/深度学习有一定了解尤佳; 4. 有GCC、LLVM和Open64等开源编译器相关开发经验尤佳; 5. 有CUDA... project... GCC/ LLVM/Open64...

30 多天前发布 · 更多.....

LLVM Senior Compiler Engineer

Byte Dance 4.1

上海市 [+2地区](#)

1. Responsible LLVM new back-end... preferred: (1) Familiar with LLVM compiler development experience...

30 天前发布 · 更多.....

SMTS Software Development Eng.

Xilinx 4.0

北京市

years of GCC/ LLVM/Open64 industry... frameworks Experience of TVM/MLIR/ LLVM/GCC is preferred Experience...

30 多天前发布 · 更多.....

Indeed 招聘信息截图



Light IR C++库

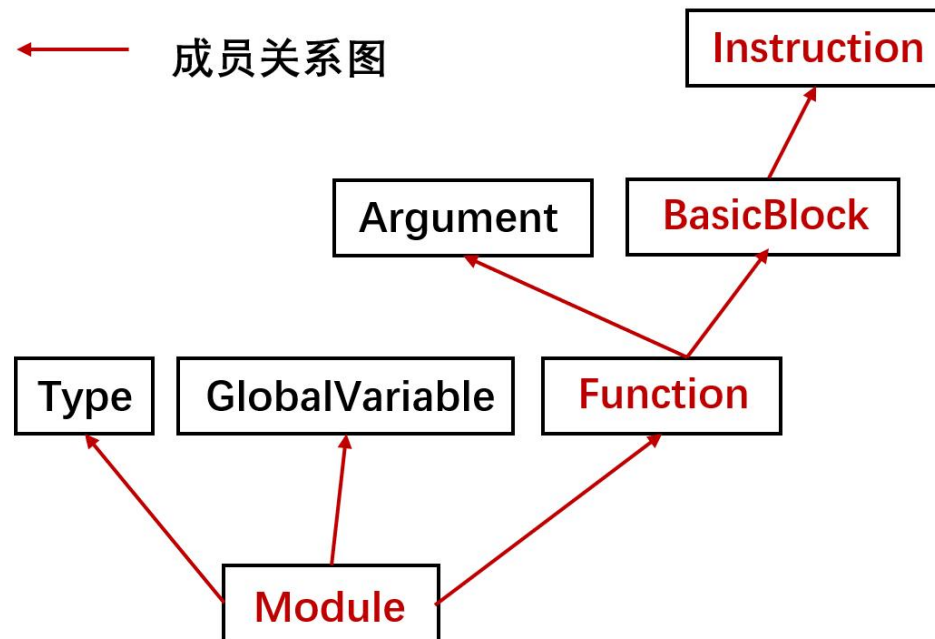
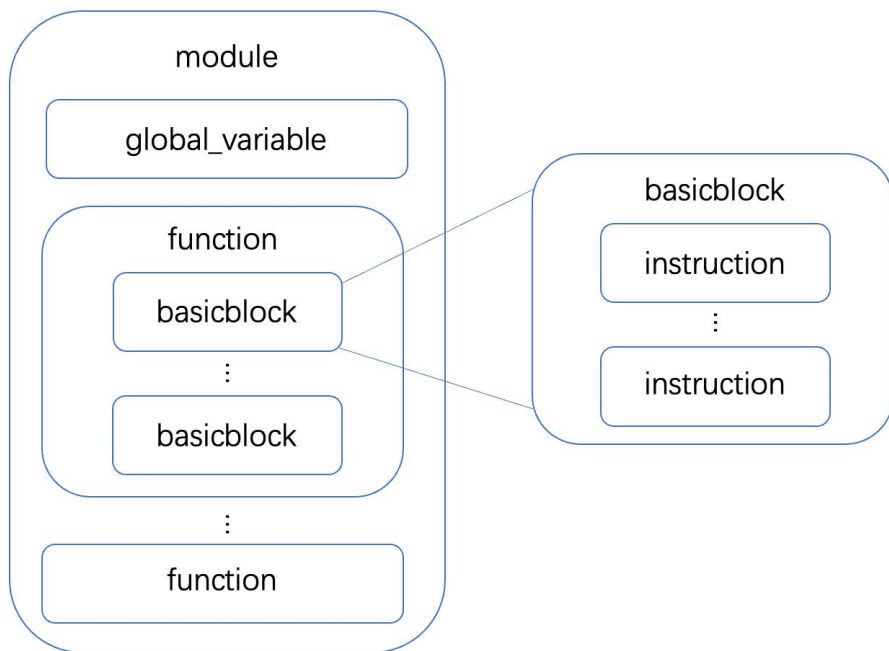
编译原理课程组

中国科学技术大学

Light IR 结构与 C++ 类总览

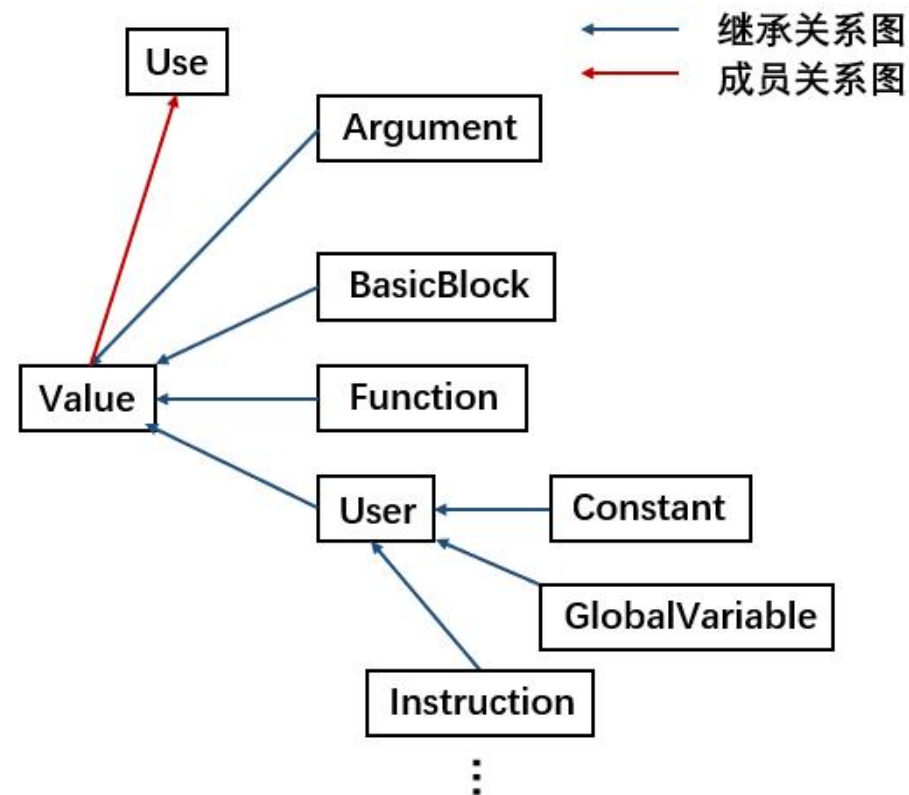


- Light IR 存在层次化结构
- Light IR C++ 库存在对应层次化类的设计

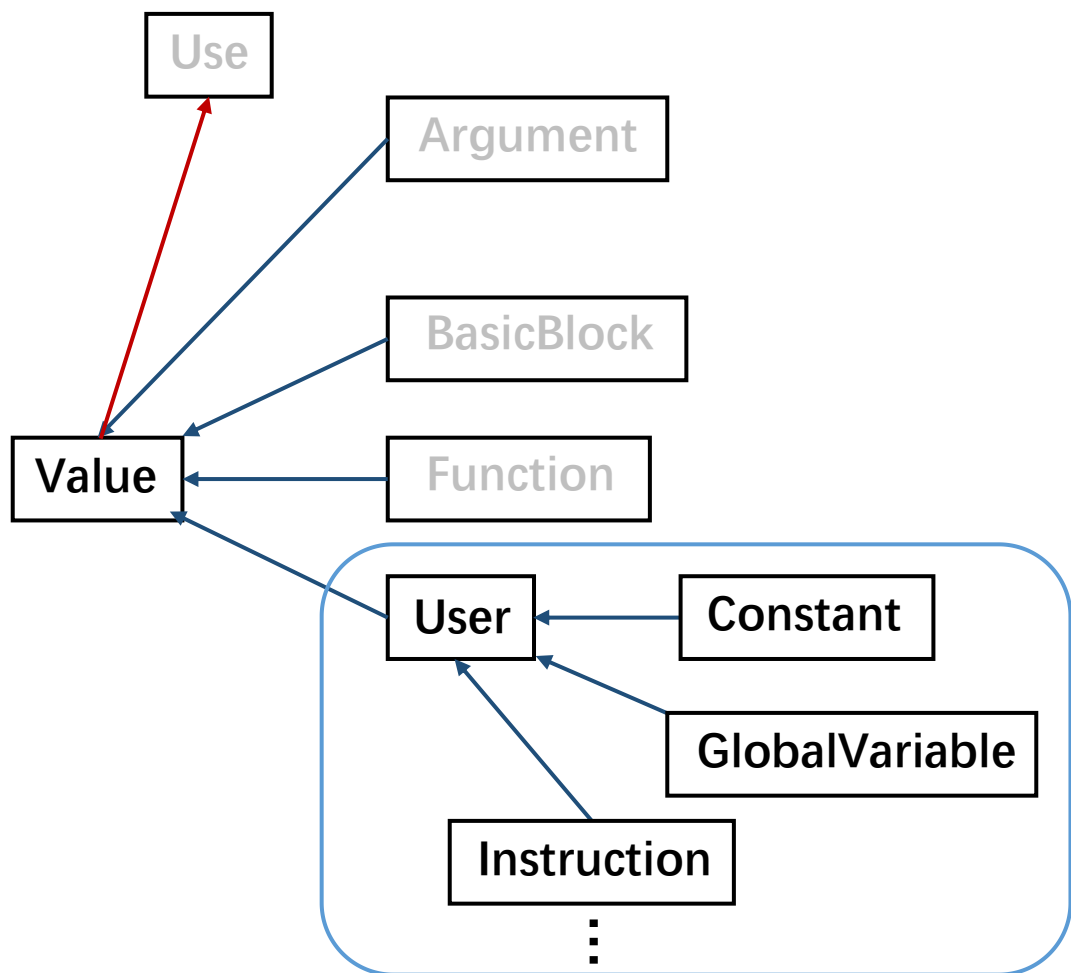


• Value

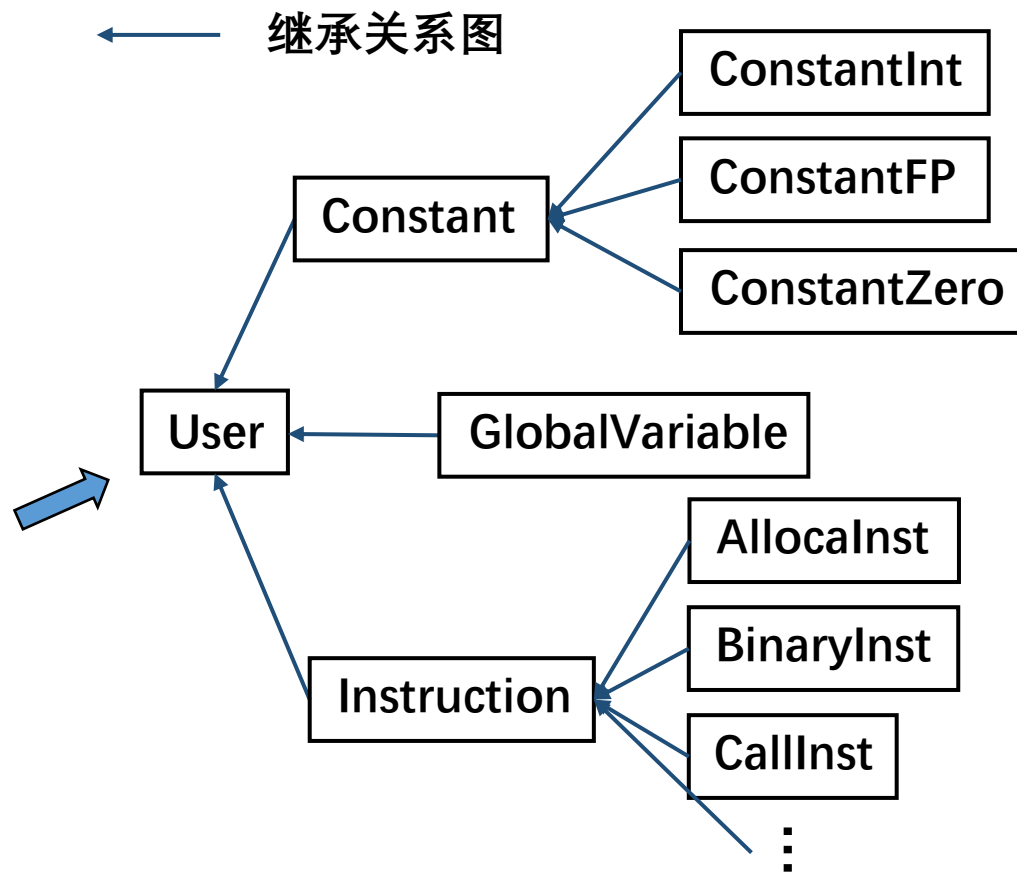
- **概念**: Value 类代表一个可用于指令操作数的带类型的数据, 包含众多子类
- **子类**: Instruction 也是其子类之一, 表示指令在创建后可以作为另一条指令的操作数
- **注**: Value 成员 use list 是 Use 类的列表, 每个 Use 类记录了该 Value 的一次被使用的情况



Light IR 数据基类

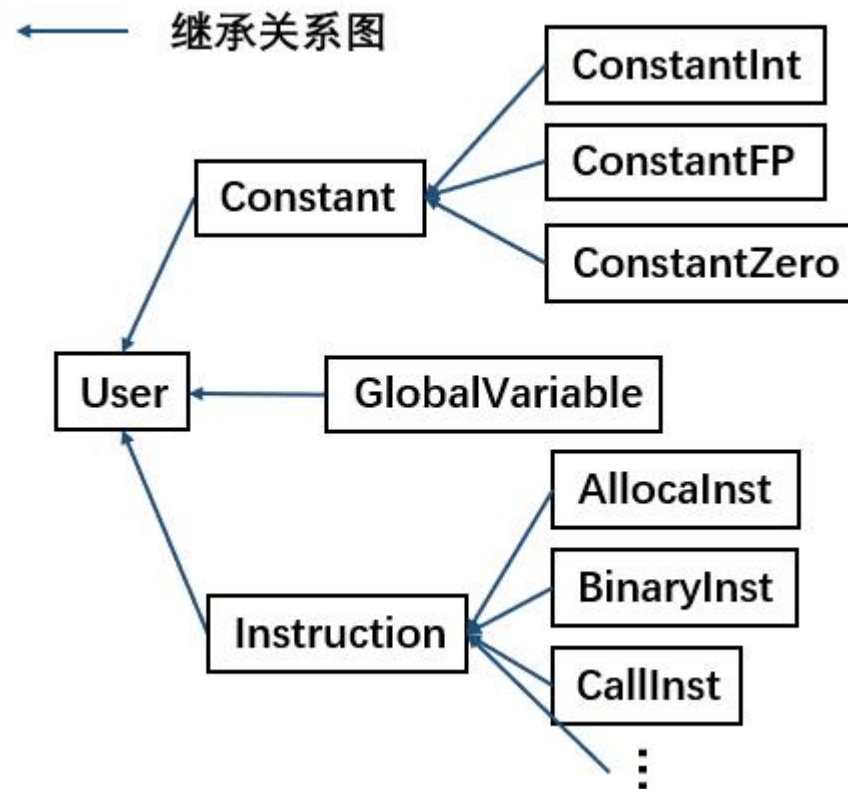


继承关系图



• User

- **概念**: User 作为 Value 的子类, 含义是使用者
- **子类**: Instruction 也是其子类之一
- **注**: User 类成员 operands 是 Value 类的列表, 表示该使用者使用的操作数列表



- **Light IR 类型系统**

- 包含基本类型与组合类型，Type 类是所有类型的基类

- **Type**

- 子类:

- IntegerType, FloatType 对应表示 Light IR 中的 i32, float 基本类型
- ArrayType, PointerType, FunctionType 对应表示组合类型: 数组类型, 指针类型, 函数类型

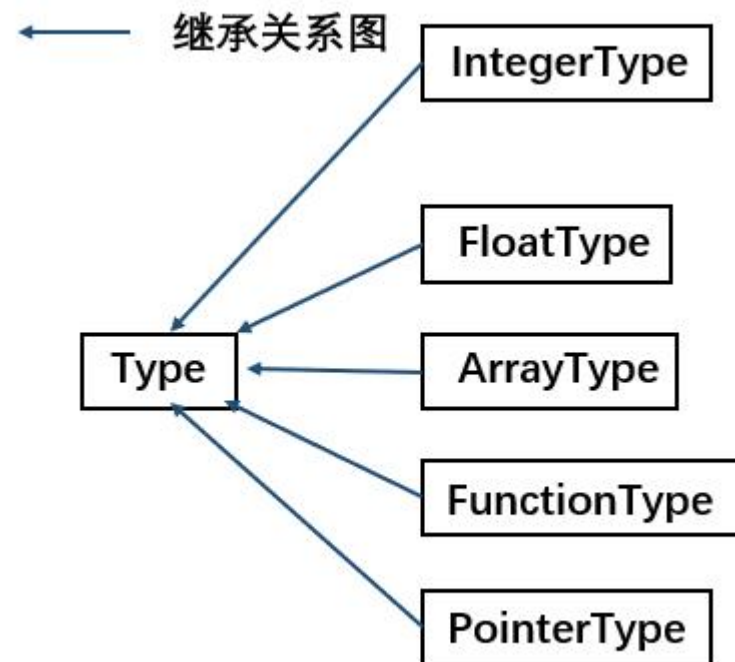
- API:

// 获取 i1 基本类型

auto int1_type = module->get_int1_type();

// 获取 [2 x i32] 数组类型

auto array_type = ArrayType::get(Int32Type, 2);



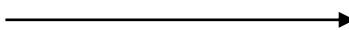
使用 Light IR C++ 库生成 IR



- 例如如下 Cminus-f 代码翻译为LightIR
- 如何用 Light IR C++库生成相应的IR?

```
int main(void) {  
    int x;  
    x = 72;  
    return x;  
}
```

Cminus-f代码




```
; ModuleID = 'cminus'  
define i32 @main() #0 {  
entry:  
    %1 = alloca i32  
    store i32 72, i32* %1  
    %2 = load i32, i32* %1  
    ret i32 %2  
}
```

翻译成 LightIR 代码

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
```



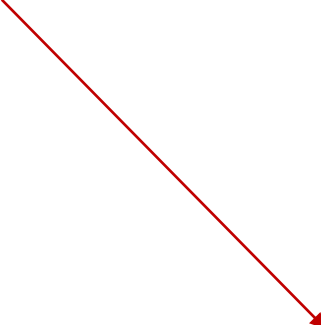
```
; ModuleID = 'cminus'
define i32 @main() #0 {
}
```

IR 情况

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
```




```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
}
```

IR 情况

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = module->get_int32_type();
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
```



```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
}
```

IR 情况

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
}
```

IR 情况

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
// 创建 load 指令, 将 x 内存值取出来
auto xLoad = builder->create_load(xAlloca);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
    %2 = load i32, i32* %1
}
```

IR 情况

使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("Cminusf code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main",
module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
// 创建 load 指令, 将 x 内存值取出来
auto xLoad = builder->create_load(xAlloca);
// 创建 ret 指令, 将 x 取出的值返回
builder->create_ret(xLoad);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
    %2 = load i32, i32* %1
    ret i32 %2
}
```

IR 情况



IR 自动化生成框架

编译原理课程组

中国科学技术大学



- 了解 Light IR **Module** 类的层次化结构, **IRBuilder** 类创建指令的流程后
- 接下来介绍框架 **CminusfBuilder** 类通过**访问者模式**遍历AST, 调用 Light IR C++ 库自动化的生成 IR

- **AST (抽象语法树) 简介**

- 分析树在 Lab1 语法分析的过程中被构造；AST 则是分析树的浓缩表示，使用运算符作为根节点和内部节点，并使用操作数作为子节点

- **Visitor Pattern (访问者模式) 概念**

- AST 类有一个方法接受访问者，将自身引用传入访问者，而访问者类中集成了对不同 AST 节点的访问规则

访问者模式示例

```
//Visitor.h  
class AddExp;  
class IntExp;  
class Visitor  
{  
public:  
    int result = 0;  
    virtual void visit(AddExp*);  
    virtual void visit(IntExp*);  
};
```

```
//Exp.h  
#include "Visitor.h"  
  
class Exp{  
public:  
    virtual void accept(Visitor&v) = 0;};  
  
class AddExp : public Exp{  
public:  
    Exp* rhs;  
    Exp* lhs;  
    AddExp(Exp* lhs, Exp* rhs) : lhs(lhs), rhs(rhs) {}  
    virtual void accept(Visitor&v) override final;};  
  
class IntExp : public Exp{  
public:  
    int value;  
    IntExp(int value) : value(value) {}  
    virtual void accept(Visitor&v) override final;};
```



访问者模式示例



```
//Visitor.cpp
#include "Exp.h"
void Visitor::visit(AddExp* add_exp){
    add_exp->lhs->accept(*this);
    add_exp->rhs->accept(*this);}

void Visitor::visit(IntExp* int_exp){
    result += int_exp->value;}

void AddExp::accept(Visitor & v){
    v.visit(this);}

void IntExp::accept(Visitor & v){
    v.visit(this);}
```

```
//main.cpp
#include <iostream>
#include <string>
#include "Exp.h"
using namespace std;

int main(){
    Exp* exp = new IntExp(1);
    for(int i = 2; i < 5; i++){
        exp = new AddExp(exp, new IntExp(i));    }

    Visitor CalSum;
    exp->accept(CalSum);
    cout << "Result is " << CalSum.result << endl;
    return 0;    }
```


访问者模式简介

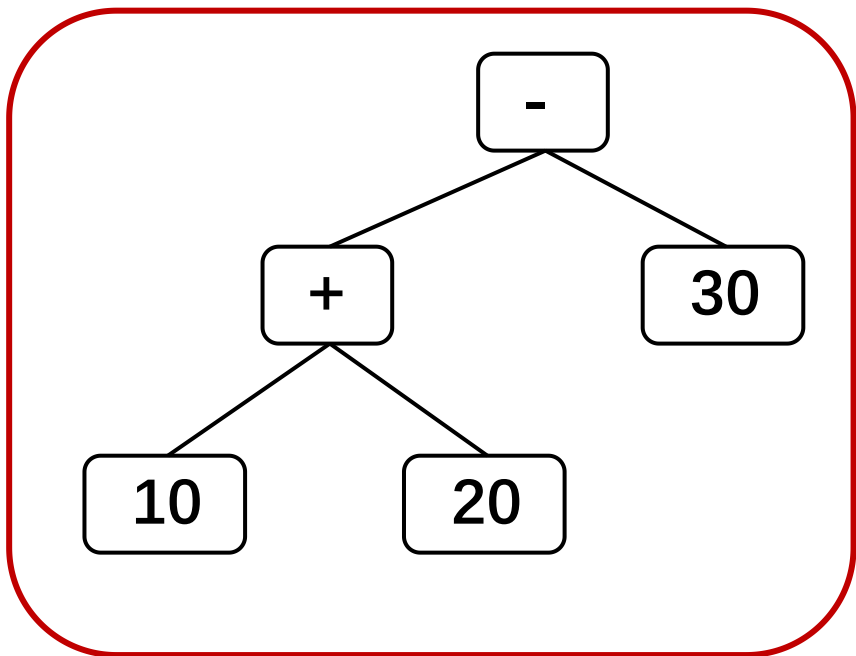


• 访问者模式例子:

- $10+20-30$

访问者类

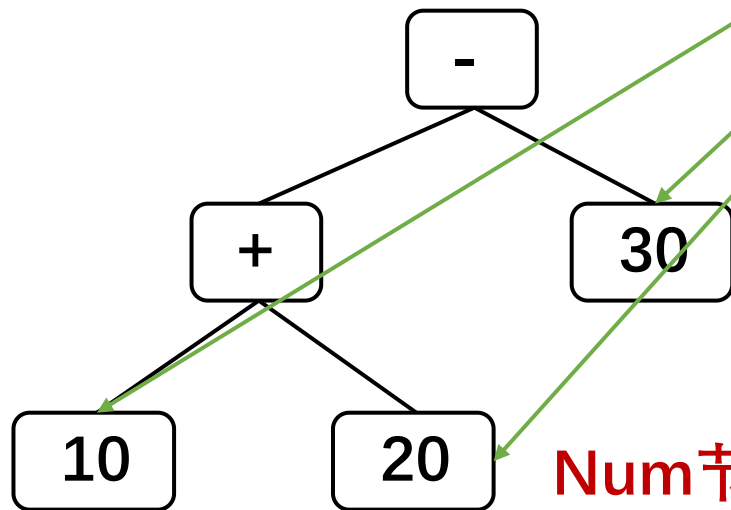
AST 结构



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

- 10+20-30

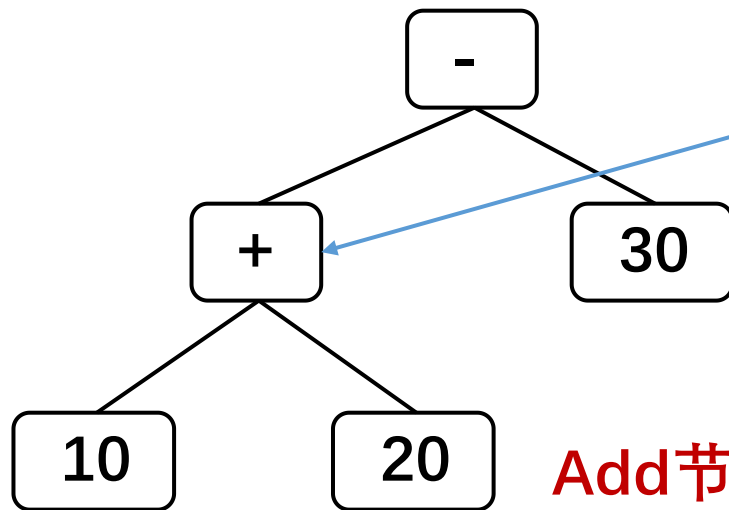


Num节点的访问规则

```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

- $10+20-30$

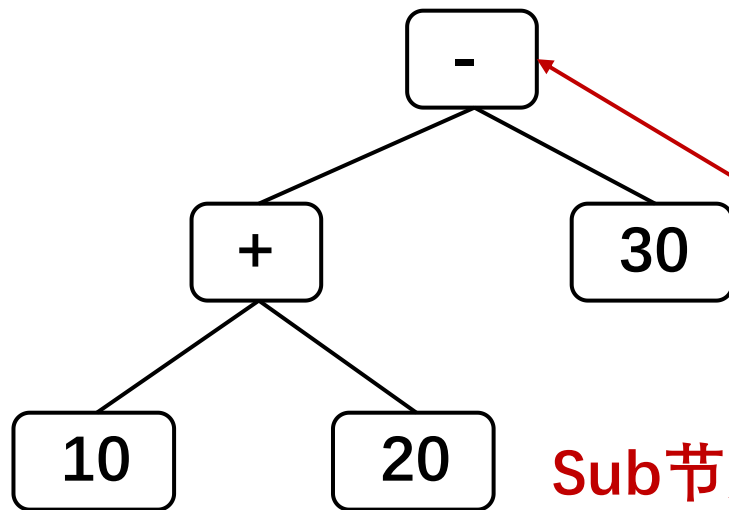


Add节点的访问规则

```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

- $10+20-30$

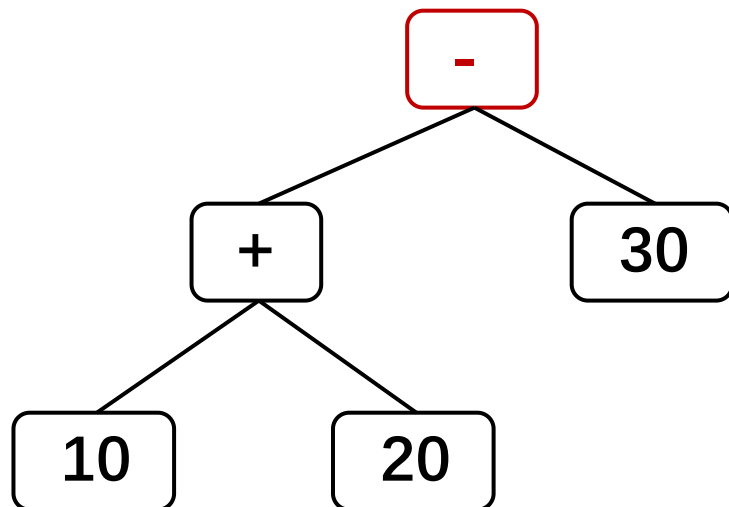


**Sub节点的访问规则，
且SubNode为根节点**

```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

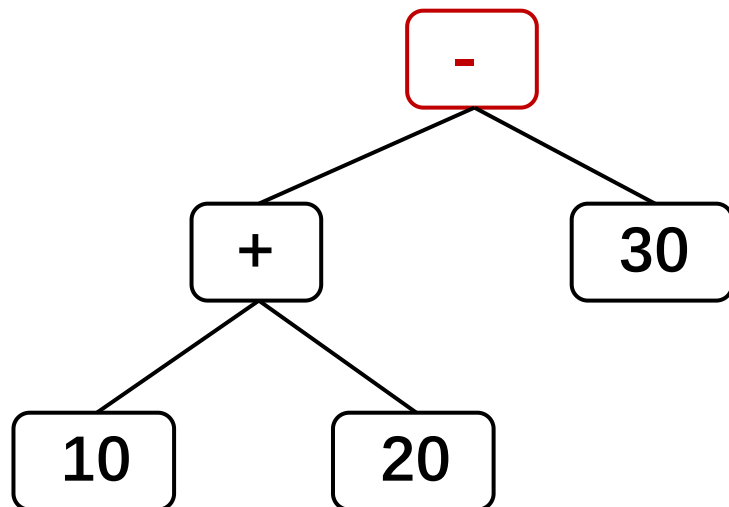
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

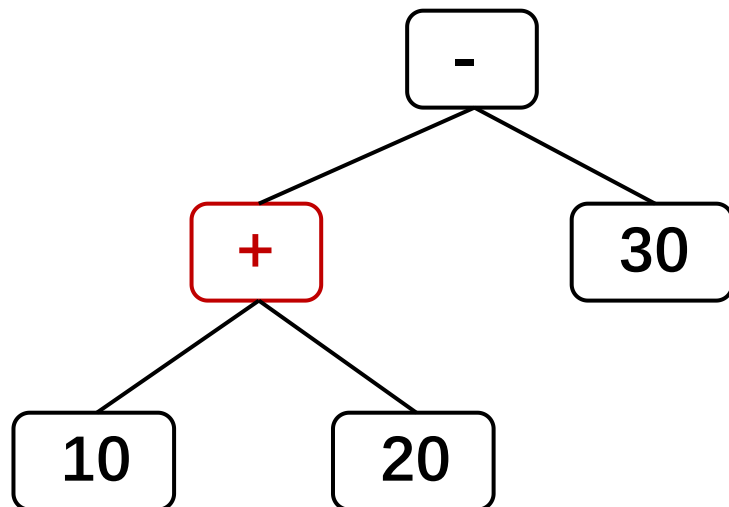
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

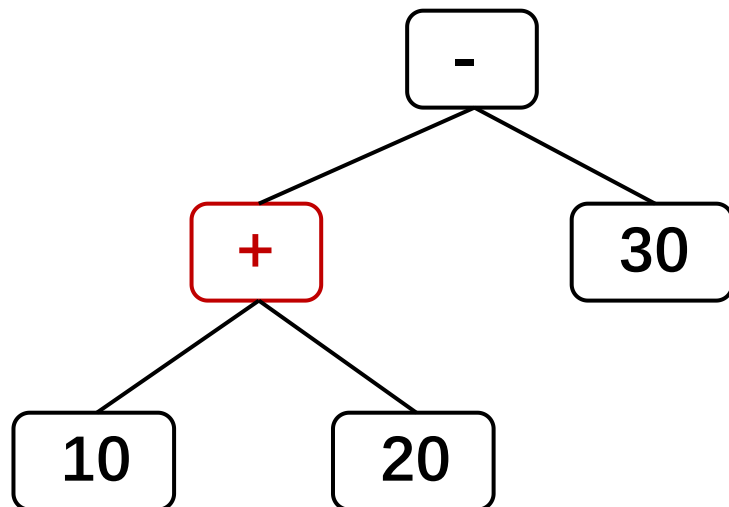
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

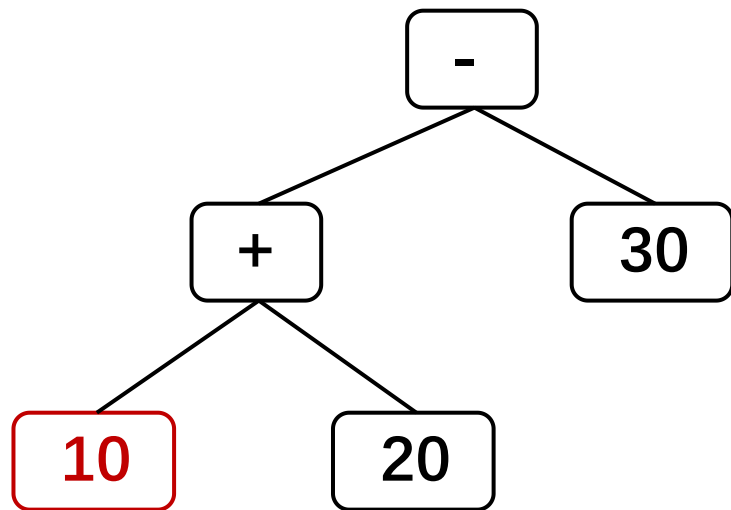
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```


• 访问者模式例子:

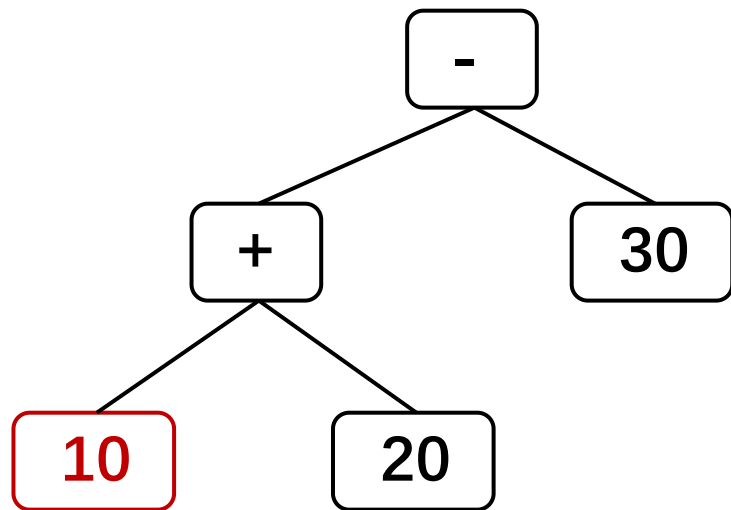
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

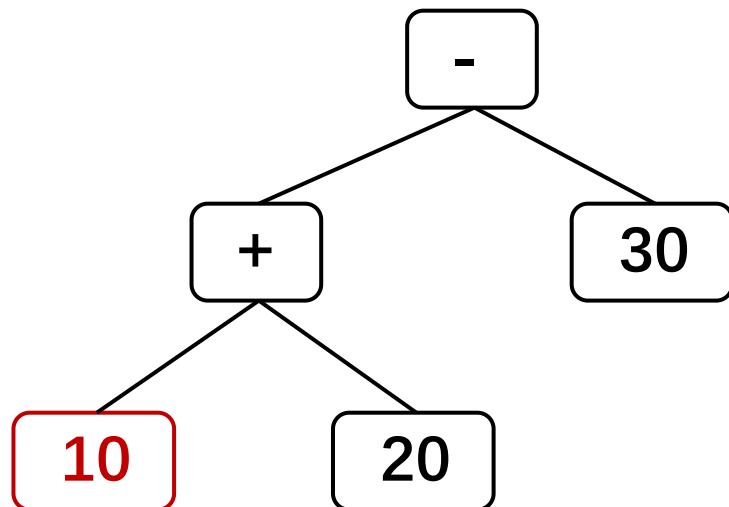
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

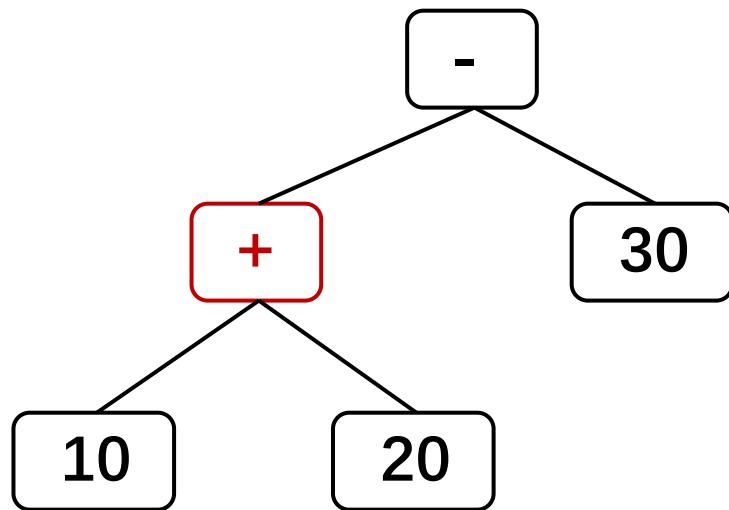
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

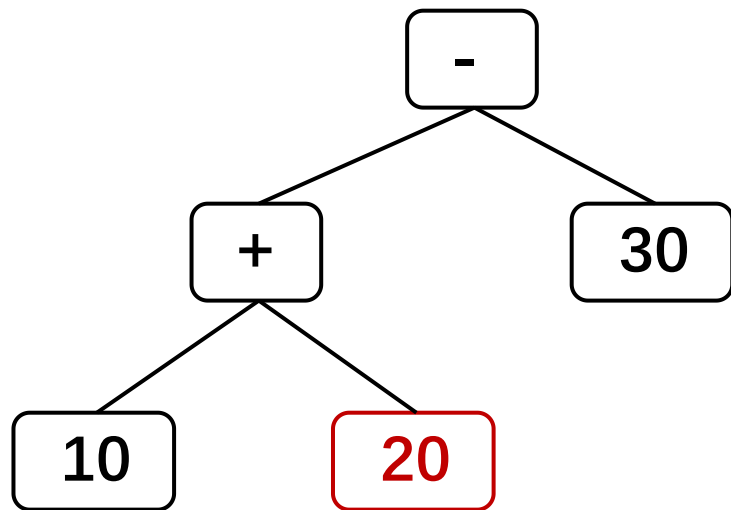
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

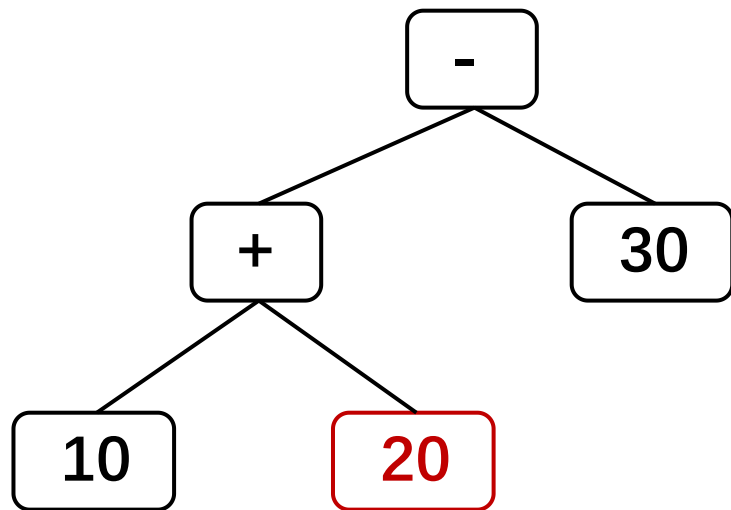
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

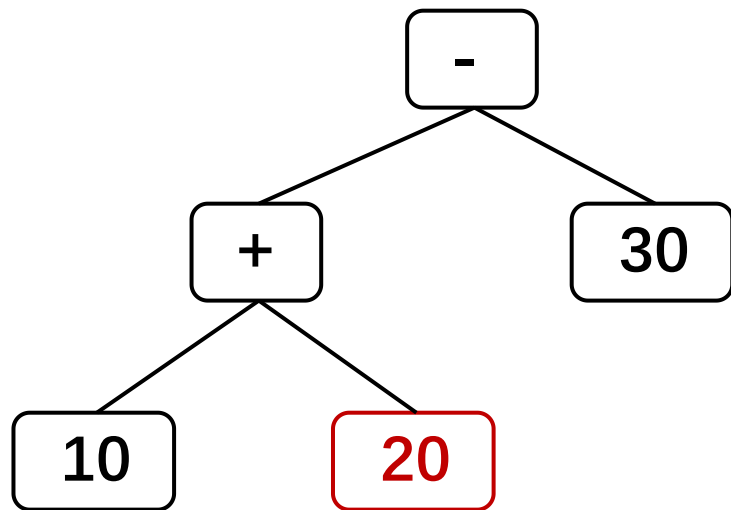
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

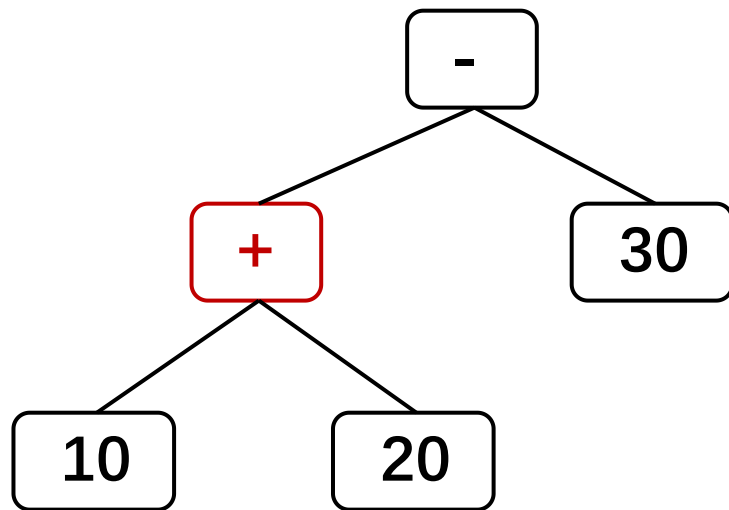
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

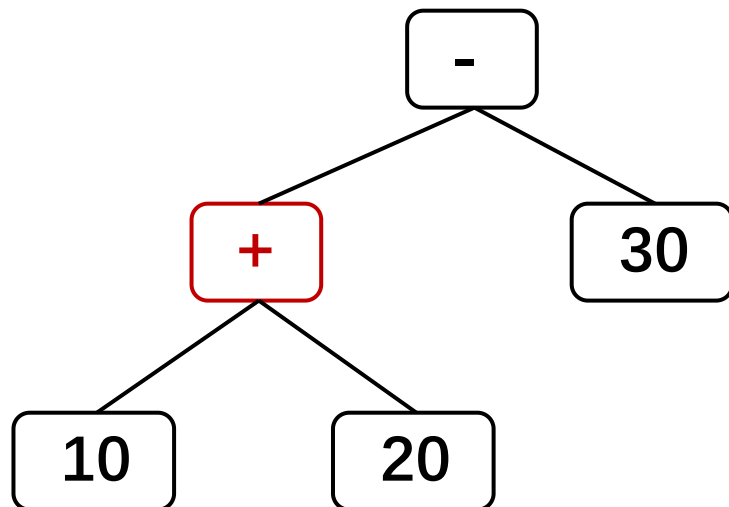
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```


• 访问者模式例子:

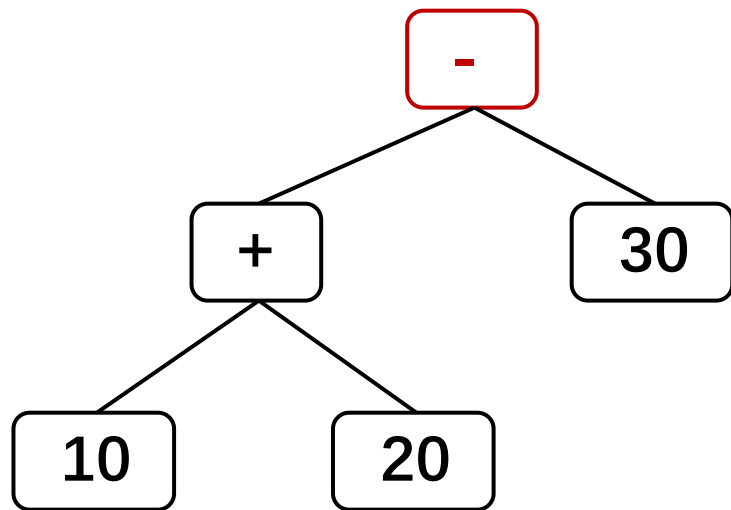
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

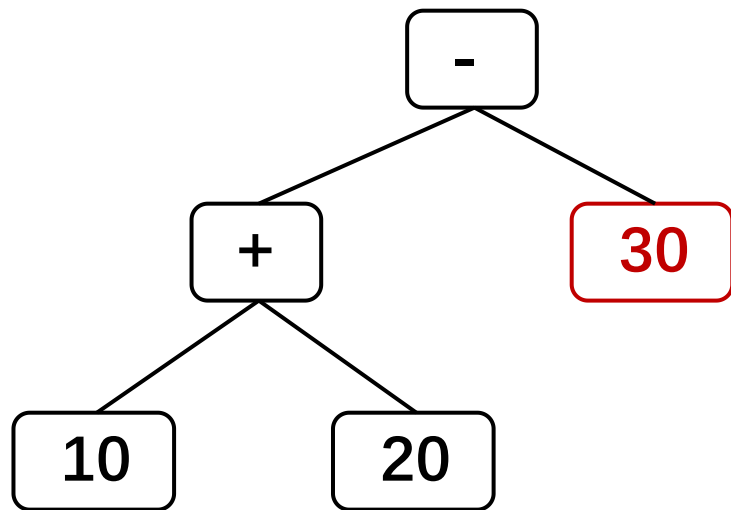
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

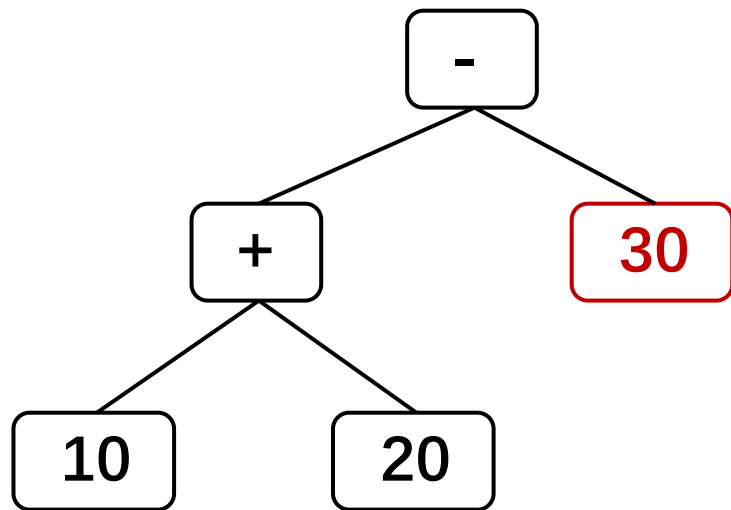
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

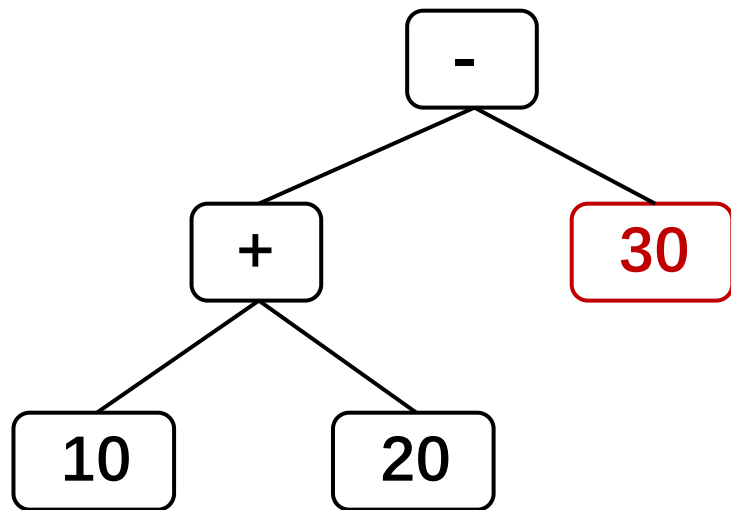
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

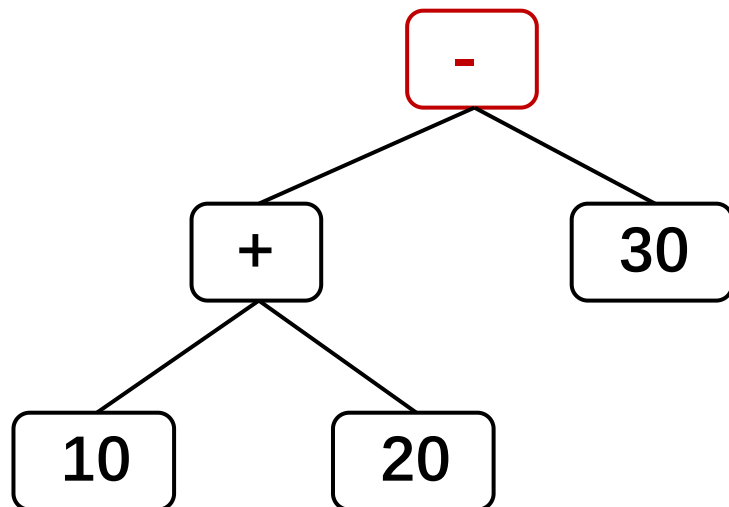
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

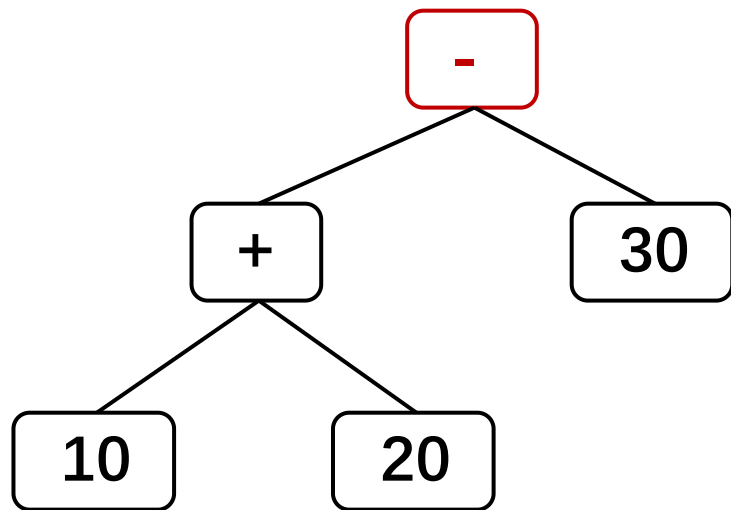
- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

• 访问者模式例子:

- $10+20-30$



```
class Calc : public ASTVisitor {
private:
    int visit(NumNode &) {
        int ret = NumNode.num;
        return ret;
    }
    int visit(AddNode &) {
        int op1 = AddNode.op1->accept(*this);
        int op2 = AddNode.op2->accept(*this);
        int ret = op1 + op2;
        return ret;
    }
    int visit(SubNode &) {
        int op1 = SubNode.op1->accept(*this);
        int op2 = SubNode.op2->accept(*this);
        int ret = op1 - op2;
        return ret;
    }
};
```

Lab 2 自动化 IR 生成



- 基于已有 AST 通过访问者模式进行 IR 自动化生成
- AST (Lab1 phase2 已实现)
- 自动化生成工具: **CminusBuilder** 类
 - 参考 `include/cminusf/cminusf_builder.hpp`
 - TODO: 需添加一些属性信息
- 访问者模式: **CminusBuilder** 中不同 **ASTnode** 的 **visit** 函数
 - `src/cminusfc/cminusf_builder.cpp`
 - TODO: visit 函数待实现

Lab 2 自动化 IR 生成



cminusfc 在开启emit-llvm选项时，会在ir构建完成后，自动调用ir的print接口，输出完整的ir代码。

AST
(Lab1 phase2 生成)

run_visitor

CminusBuilder
(通过 AST 生成 IR)

emit-llvm

Light IR code

visit ast生成IR，填充irbuilder属性 (Lab2 待实现)

IRbuilder拥有一个module类属性，需要按照 module-function-basicblock-instruction的结构去填充。

```
class CminusfBuilder : public ASTVisitor {
public:
    CminusfBuilder() {
        module = std::make_unique<Module>();
        builder
=std::make_unique<IRBuilder>(nullptr,
module.get());
        private:
            virtual Value *visit(ASTProgram &)
override final;
        ...
        std::unique_ptr<IRBuilder> builder;
        Scope scope;
        std::unique_ptr<Module> module;
        struct {} context; // TODO
    };
};
```

CminusfBuilder类：

module： 一个Module类指针，其组织结构包括其内function，function内BB，instr。也是本次是在构建builder时需要填充的核心属性

visit函数： 访问者模式下，帮助访问各类ASTnode的函数接口，这里只给出了对于ASTProgram 的visit函数

context： 构建ir中的上下文信息，可以当作全局变量使用，在实现时根据需要补充。

AST->lightIR 转换示例



Cminusf源文件

```
int main(void)
{
    return 0;
}
```

emit-ast

AST实例

```
program
--fun-declaration: main
----compound-stmt
-----return-stmt
-----simple-expression
-----additive-expression
-----term
-----num (int): 0
```

emit-llvm

lightir 源文件

```
define i32 @main() {
label_entry:
    ret i32 0
}
```



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

2024年10月12日