中国科学技术大学
University of Science and Technology of China

程序设计II

第8讲 从C到C++

计算机学院 黄章进

zhuang@ustc.edu.cn

- C程序设计实践
  - 字符串处理
  - 高精度计算
  - 枚举
  - 递归
- C++语言
  - 类和对象
  - 运算符重载
  - 继承与派生
  - 虚函数与多态

内容

- 8.1 面向过程与面向对象
- 8.2 面向对象的基本概念
- 8.3 C++面向过程的扩充

C的不足

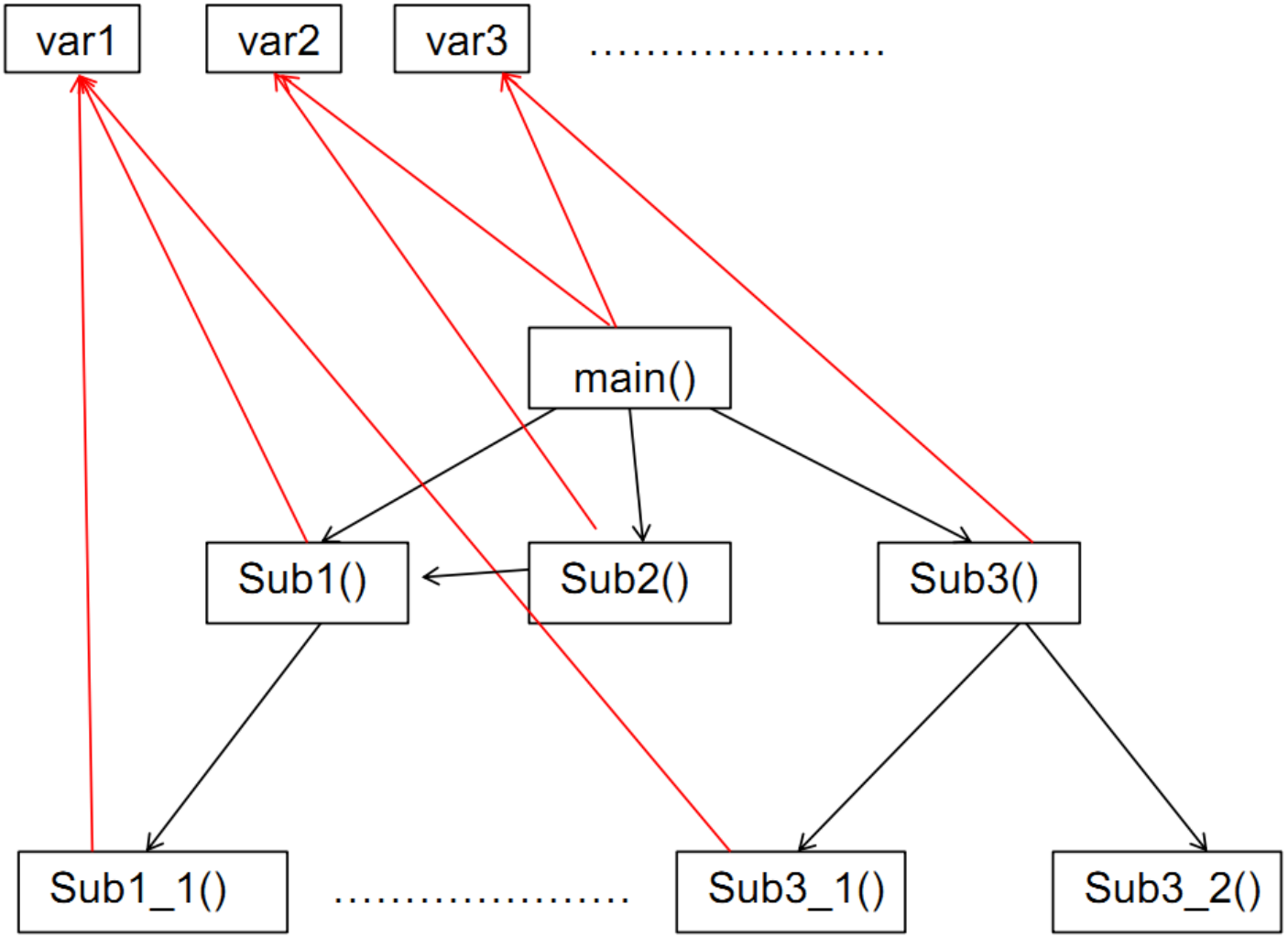
- C语言是面向过程的结构化程序设计语言
  - C语言不是专门用来开发大型程序的
    - UNIX系统的内核大约是10000行C代码（加上小部分汇编代码）
- C++是面向对象的程序设计语言
  - 面向对象的程序设计(OOP)思想是针对开发较大规模程序而提出的
    - 更直接地描述客观世界中存在的事物(对象)以及它们之间的关系

面向过程的结构化程序设计

程序 = 算法 + 数据结构

- 程序由全局变量以及众多相互调用的函数组成
- 算法以函数的形式实现，用于对数据结构进行操作
- 算法和数据结构是互相独立、分开设计的，以算法为主体
- 设计思想：自顶向下，逐步求精；模块分解，分而治之

面向过程的结构化程序设计



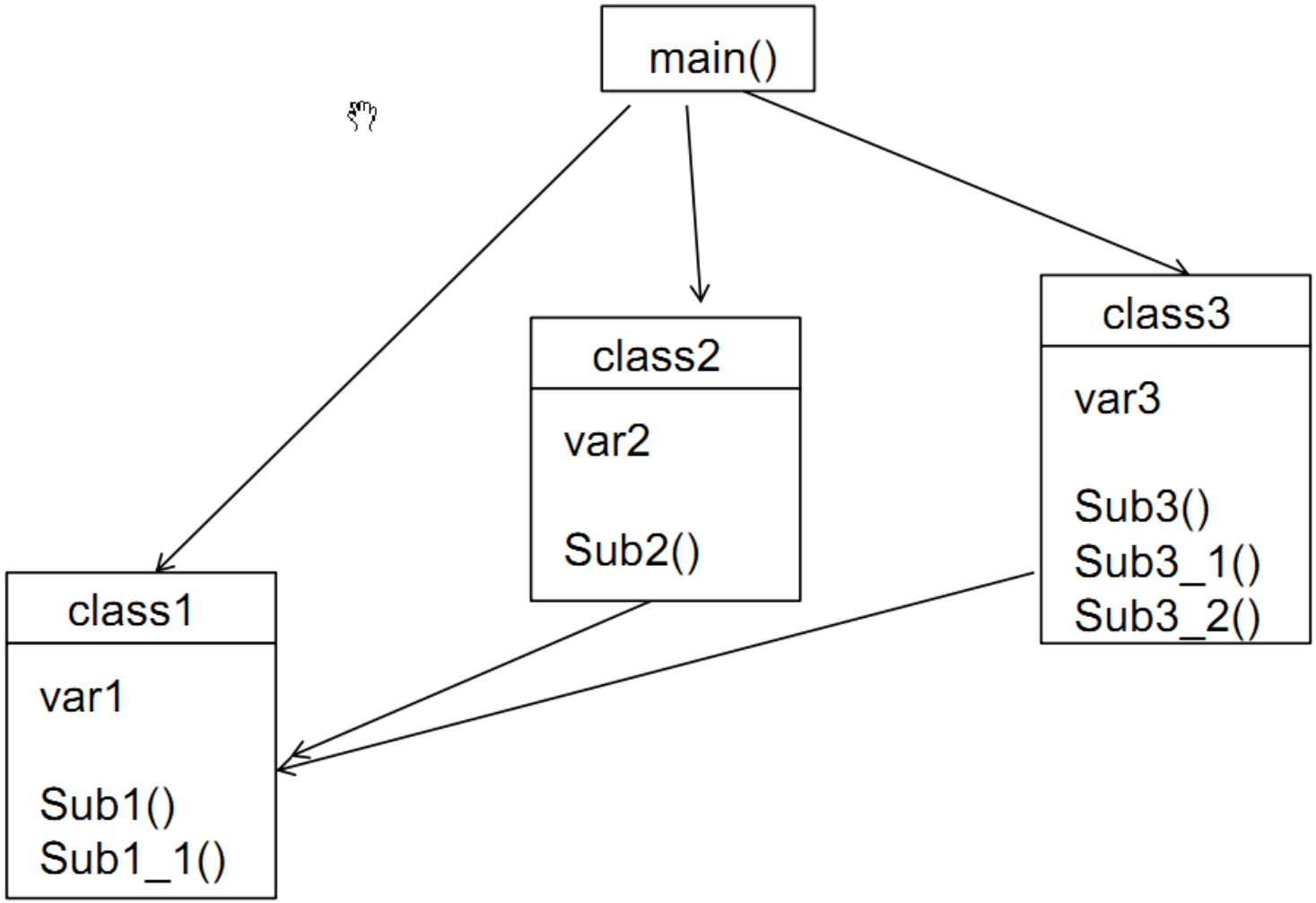
面向过程的结构化程序设计

- 优点
  - 直观有条理、结构清晰、模块化强
- 缺点
  - 数据结构对整个程序公开，数据安全性差
  - 代码的可重用性差、模块之间依赖性强、不利于代码的维护和扩充
    - 数据与处理数据的过程相分离

面向对象的程序设计

- 将数据及对数据的操作方法封装在一起，作为一个相互依存、不可分离的整体——对象
- 对同类型对象抽象出其共性，形成类
- 类中的大多数数据，只能用本类的方法进行处理
  - 类通过一个简单的外部接口，与外界发生关系
- 对象与对象之间通过消息进行通信

面向对象的程序设计


main()
class1
var1
Sub1()
Sub1_1()
class2
var2
Sub2()
class3
var3
Sub3()
Sub3_1()
Sub3_2()

面向对象的程序设计

程序 = 对象s + 消息

类/对象 = 算法 + 数据结构

- 优点
  - 程序模块间的关系更为简单，程序模块的独立性、数据的安全性就有了良好的保障。
  - 通过继承与多态性，可以大大提高程序的可重用性，使得软件的开发和维护都更为方便。
- 缺点
  - 代码的效率有时会略低

8.2面向对象的基本概念

- 对象
- 类
- 封装
- 继承
- 多态

对象

- 一般意义上的对象：
  - 是现实世界中一个实际存在的事物。
  - 可以是有形的（比如一辆汽车），也可以是无形的（比如一项计划）。
  - 是构成世界的一个独立单位，具有
    - 静态特征：可以用某种数据来描述
    - 动态特征：对象所表现的行为或具有的功能

对象

- 面向对象方法中的对象：
  - 是系统中用来描述客观事物的一个实体，它是用来构成系统的一个基本单位。
  - 对象由一组属性和一组行为构成。
    - 属性：用来描述对象静态特征的数据项。
    - 行为：用来描述对象动态特征的操作序列

类

- 分类——人类通常的思维方法
- 分类所依据的原则——抽象
  - 忽略事物的非本质特征，只注意那些与当前目标有关的本质特征，从而找出事物的共性，把具有共同性质的事物划分为一类，得出一个抽象的概念。
  - 例如，石头、树木、汽车、房屋等都是人们在长期的生产和生活实践中抽象出的概念。

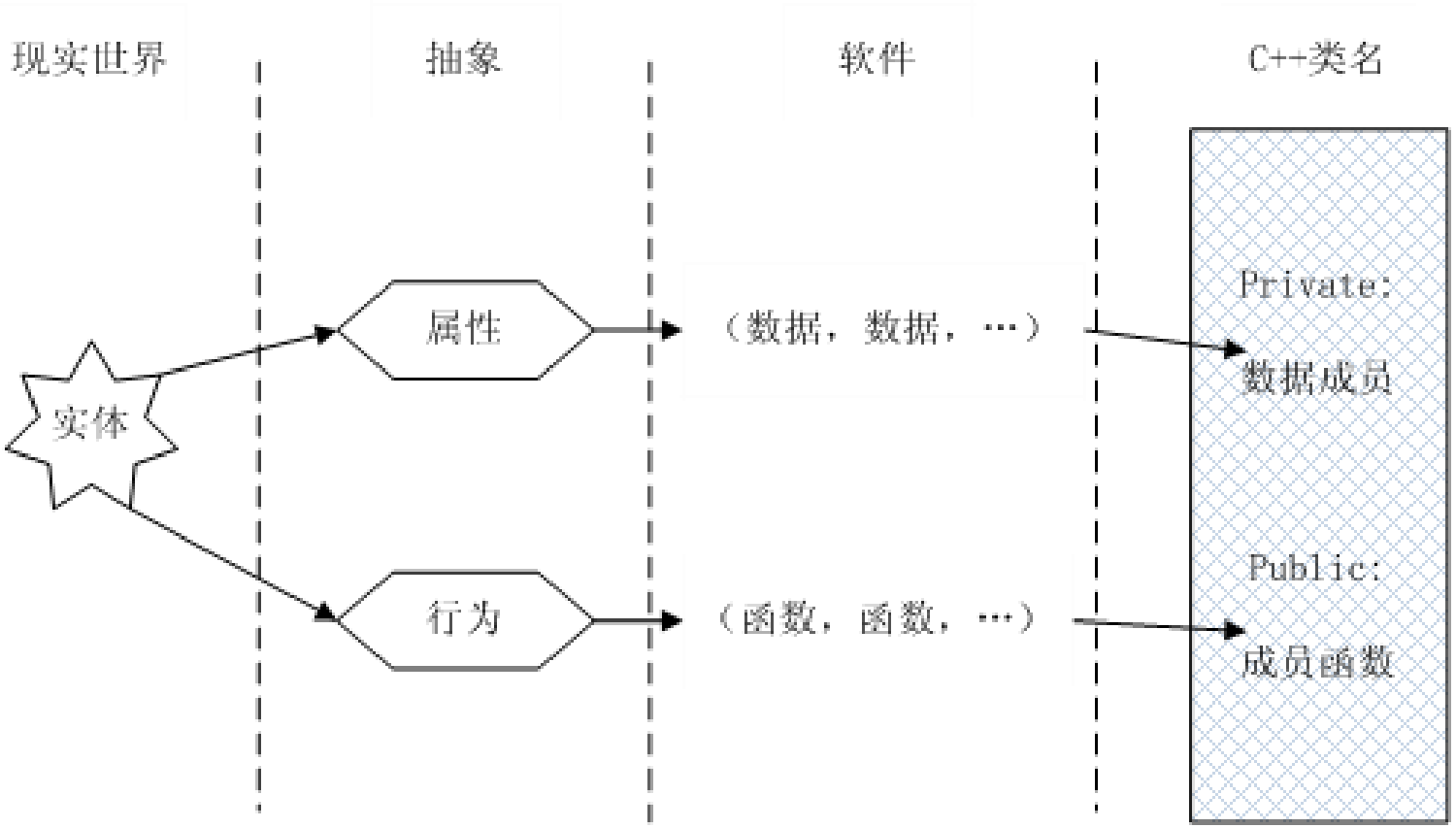
类

- 类是具有相同属性和操作的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述。
- 对象是对问题域中客观存在事物的抽象，是类的具体的个体，也称为类的一个实例
- 类与对象是一对抽象与具体的关系
  - 相当于类型和变量的关系

封装

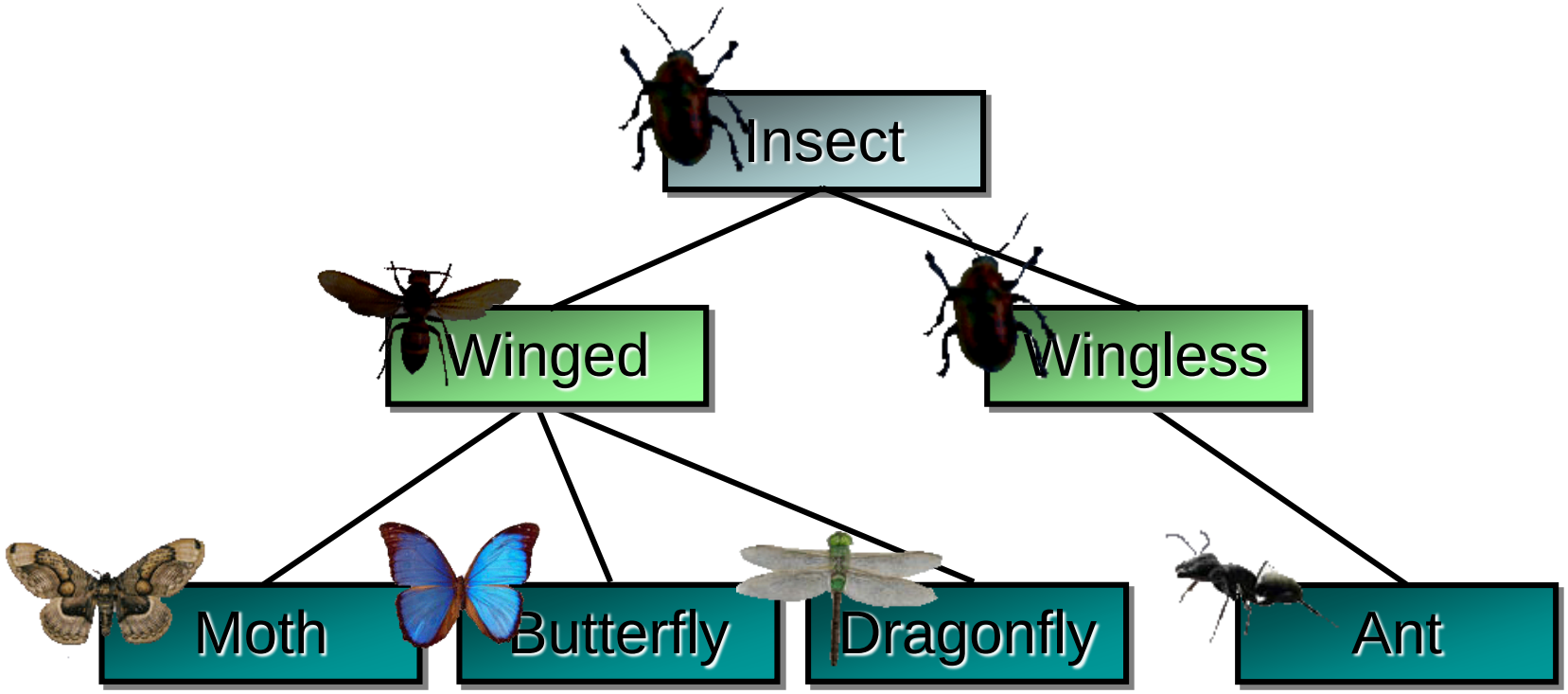
- 将一组数据和这组数据相关的操作集合封装在一个对象中，形成一个基本单元
- 信息隐藏：尽可能隐蔽对象的内部细节，只保留有限的对外接口使之与外部发生联系。
- 在C++中，实现数据封装的机制是“类(class)”

封装



继承

- 在客观世界中，存在着一般和特殊的关系(is a)



继承

- 继承对于软件复用有着重要意义，是面向对象技术能够提高软件开发效率的重要原因之一。
- 在面向对象的语言中，类功能支持继承机制。
  - 继承使一个类（称为基类或父类）的数据和操作能被另一个类（称为派生类或子类）重用
  - 一个子类从它的父类那里继承了所有的数据和操作，并扩充自己的特殊数据和操作。
  - 父类抽象出共同特征，子类表达差别。

多态

- 一种接口，多种形态
  - 发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。
- 静态多态（编译时多态）
  - 函数重载
  - 运算符重载
- 动态多态（运行时多态）
  - 虚函数

```
void fun(int, int, char);
void fun(char, float);
void fun(int, int);
void fun(float, float);
```

8.3 C++面向过程的扩充

- 简单C++程序
- 名字空间
- 输入输出
- bool类型
- 引用
- 显式类型转换
- 函数

参考书

- Stanley B. Lippman, et al. *C++ Primer (4th Edition)*, Addison-Wesley Professional, 2005.
  - C++Primer中文版(第4版), 李师贤 等译, 人民邮电出版社, 2008.
- Bjarne Stroustrup. *The C++ Programming Language: Special Edition*, Addison-Wesley Professional, 2000
  - C++程序设计语言(特别版 十周年中文纪念版)，裘宗燕 译, 机械工业出版社, 2010.
- Bruce Eckel. *Thinking in C++(2nd Edition)*, Prentice Hall, 2000.
  - C++编程思想, 刘宗田 等译, 机械工业出版社.

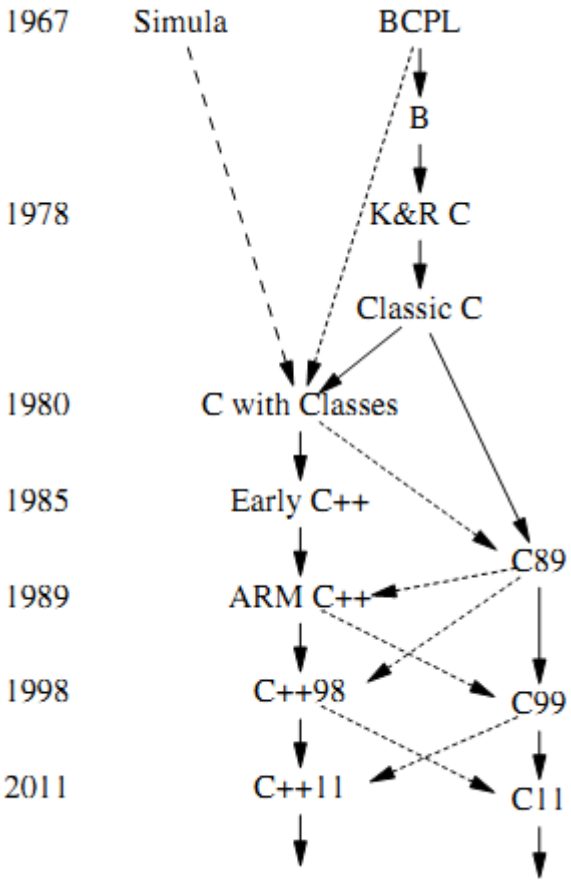
参考书 (C++11)

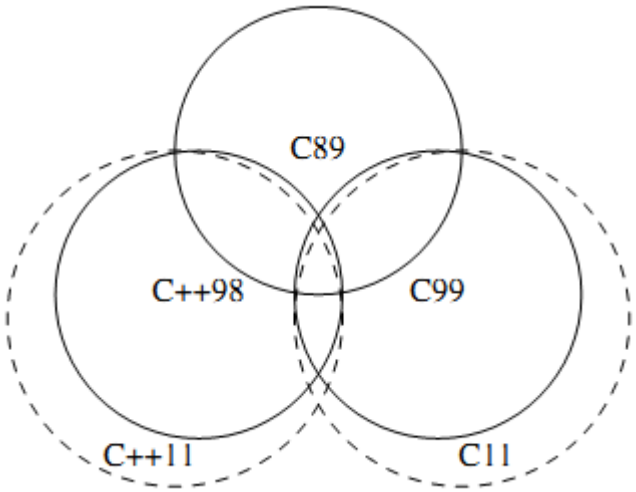
- Stanley B. Lippman, et al. *C++ Primer (5th Edition)*, Addison-Wesley Professional, 2012.
  - C++Primer中文版(第5版), 王刚 等译, 电子工业出版社, 2013.
- Bjarne Stroustrup. *The C++ Programming Language: 4th Edition*, Addison-Wesley Professional, 2013
  - A Tour of C++, Addison-Wesley Professional, 192 pages, 2013

C++语言的产生

- C++是从C语言发展演变而来的，首先是一个更好的C
- 引入了类的机制，最初的C++被称为“带类的C”
- 1983年正式取名为C++
- 从1989年开始C++语言的标准化工作
- 于1994年制定了ANSI C++标准草案
- 于1998年11月被国际标准化组织（ISO）批准为国际标准，第一版的C++98(C++03)
- 于2011年8月发布了第二版的ISO C++标准：C++11(C++14)

C++和C是兄弟


1967
Simula
BCPL
B
1978
K&R C
Classic C
1980
C with Classes
1985
Early C++
C89
1989
ARM C++
1998
C++98
C99
2011
C++11
C11


C89
C++98
C99
C++11
C11

C++的特点

- 兼容C
  - 它保持了C的简洁、高效和接近汇编语言等特点
  - 对C的类型系统进行了改革和扩充
  - C++也支持面向过程的程序设计，不是一个纯正的面向对象的语言
- 支持面向对象的方法

简单C++程序


```
#include <iostream> //包含ISO C++头文件
using namespace std; //使用预定义的名字空间std

int main(void)
{
    cout << “Hello, world!” << endl;  //在屏幕上输出字符串
    return 0;
}
```

hello.cpp

ISO C++库文件

#include <iostream>

- iostream是ISO C++库文件，不带有后缀.h
- 很多标准C++函数继承自标准C，头文件以c开头
  - cmath，cstdlib，cstdio，ctime……

名字空间

using namespace std;

- 包含头文件iostream的目的是为了使用标准输出流cout。
- 但如果仅仅是在原代码中包含了iostream，程序还是不能正确地通过编译，因为编译器不认识cout和endl这两个标识符。所以，我们通过using namespace std语句来引入名为std的名字空间。

名字空间

- 名字空间可以解决类名、函数名等的命名冲突
- 名字空间的声明

```
namespace 名字空间名 {
    各种声明（函数声明、类声明、……）
}
```

- 例

```
namespace SomeNs {
    class SomeClass { ... };
}
```

名字空间

- 名只空间允许嵌套

```
namespace OuterNs {
  namespace InnerNs {
      class SomeClass { ... };
  }
}
```

- 特殊的名字空间
  - 全局名字空间：默认的名字空间
    - 在显式声明的名字空间外声明的标识符都在全局名字空间中
  - 匿名名字空间：对每个源文件是唯一的

名字空间作用域

- 一个名字空间确定了一个名字空间作用域
- 引用其它名字空间作用域中的标识符

  名字空间名::标识符名

  – 例：声明一个SomeClass型的对象

  ```
  SomeNs::SomeClass obj1;
  OuterNs::InnerNs::SomeClass obj2;
  ```

- 将其它名字空间作用域的标识符暴露于当前作用域

  – 对指定标识符

  ```
  using 名字空间名::标识符名;
  ```

  – 对所有标识符

  ```
  using namespace 名字空间名;
  ```

命名空间的using声明

- 作用域操作符(::)的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧名字
  - std::cin表示使用命名空间std中的名字cin
- 使用using声明(using declaration)后无需namespace_name::前缀就能直接访问命名空间中的名字

  using *namespace*::*name*;
- 一般地，头文件不应包含using声明，以避免名字冲突

多个名字的using声明

- 每条using声明引入命名空间的一个成员，且都得以分号结束

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

输入输出

cout << “Hello, world!” << endl;

- 在C++中，使用标准输出流对象cout进行输出
  - “<<” 插入运算符，重载的左移位运算符
  - 控制符endl代表回车换行操作，作用与"\n"相同
  - 格式为：

  cout << 表达式1 << 表达式2…;

表 1.2　C++ 预定义的标准流

流名	含义	隐含设备	流名	含义	隐含设备
cin	标准输入	键盘	cerr	标准出错输出	屏幕
cout	标准输出	屏幕	clog	cerr 的缓冲形式	屏幕

输入输出

- 使用标准输入流对象cin进行输入
  - 格式为

    cin >> 变量名1 >> 变量名2…;
  - ">>" 提取运算符，重载的右移位运算符
- 格式控制

输入输出


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

int main(void)
{
    int a, b;
    cin >> a >> b;
    cout << "a + b = " << a + b <<
endl;
    return 0;
}
```

```
#include <cstdio>

int main(void)
{
    int a, b;
    scanf("%d %d", &a, &b);
    printf("a + b =  %d\n", a+b);
    return 0;
}
```

bool类型

- C99通过在头文件stdbool.h中定义bool, true, false宏
- C++增加了数据类型bool，用常量true表示逻辑真，false表示逻辑假

引用

- 引用(reference)为对象取一个别名
  - 声明符写成&*d*的形式，其中*d*是声明的变量名
  - 定义引用时，必须初始化，绑定到一个已存在的对象
  - 一旦初始化完成，无法令引用绑定到另外一个对象

```
int ival = 1024;
int &refVal = ival;  // refVal refers to (is another name for) ival
int &refVal2;        // error: a reference must be initialized
```

引用即别名

- 对引用的所有操作都是在与之绑定的对象上进行的
  - 为引用赋值，实际上是把值赋给与引用绑定的对象
  - 获取引用的值，实际上是获取与之绑定的对象的值
  - 以引用作为初始值，实际上是以与引用绑定的对象作为初始值
- 引用本身不是一个对象，因此不能定义引用的引用

引用即别名


```
int ival = 1024;
int &refVal = ival;  // refVal refers to (is another name for) ival
refVal = 2;      // assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal;  // same as ii = ival

// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;
// initializes i from the value in the object to which refVal is bound
int i = refVal; // ok: initializes i to the same value as ival
```

引用的定义

- 允许在一条语句中定义多个引用，但每个引用标识符都必须以符号&开头

```
int i = 1024, i2 = 2048;  // i and i2 are both ints
int &r = i, r2 = i2;      // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3;  // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2;   // both r3 and r4 are references
```

引用的定义

- 一般地，引用的类型要和与之绑定的对象严格匹配
- 一般地，引用只能绑定到对象，不能与字面值或表达式的计算结果绑定

```
int &refVal4 = 10;   // error: initializer must be an object
double dval = 3.14;
int &refVal5 = dval; // error: initializer must be an int object
```

练习

- 练习2.15：下面哪些定义是不合法的？

  (a) int ival = 1.01;

  (b) int &rval1 = 1.01;

  (c) int &rval2 = ival;

  (d) int &rval3;

- 练习2.17：读代码写结果

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

练习

- 练习2.16：哪些赋值是不合法的，如果合法，执行了什么样的操作？

  int i = 0, &r1 = i; double d = 0, &r2 = d;

  (a) r2 = 3.14159;

  (b) r2 = r1;

  (c) i = r2;

  (d) r1 = d;

指针的引用

- 引用本身不是一个对象，不能定义指向引用的指针
- 指针是对象，可以定义指针的引用(refenrence to pointer)

```
int i = 42;
int *p;          // p is a pointer to int
int *&r = p;   // r is a reference to the pointer p
r = &i; //  r refers to a pointer; assigning &i to r makes p point to i
*r = 0; //  dereferencing r yields i, the object to which p points; changes i to 0
```

指针的引用


```
int *&r = p;   // r is a reference to the pointer p
```

- 要理解r的类型，可以从右向左阅读r的定义
  - 离变量名最近的符号(&)对变量的类型有最直接的影响，因此r是一个引用
  - 声明符的其余部分用以确定r引用对象的类型，符号*说明r引用的是一个指针
  - 声明的基本类型部分指出r引用的是一个int型指针

const限定符

- const限定的变量在运行时不能改变其值（即，运行时只读常量）
- const对象创建后其值就不能改变，所以必须初始化
  - 初始值可以是任意复杂的表达式

```
const int bufSize = 512;    // input buffer size
bufSize = 512; // error: attempt to write to const object
const int i = get_size();  // ok: initialized at run time
const int j = 42;          // ok: initialized at compile time
const int k;               // error: k is uninitialized const
```

const对象仅在文件内有效

const限定符

- 默认情况下，const对象被设定为仅在文件内有效
  - 多个文件中出现了同名的const变量时，等同于在不同文件中分别定义了独立的变量
- 如果想在多个文件间共享const对象，必须在变量的定义和声明前加extern

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

练习

const限定符

◆练习2.26：下面哪些语句是合法的？

(a) const int buf;

(b) int cnt = 0;

(c) const int sz = cnt;

(d) ++cnt; ++sz;

const的引用

const限定符

- 可以把引用绑定到const对象上，称之为对常量的引用(reference to const)
- 不能通过对常量的引用修改它所绑定的对象

```
const int ci = 1024;
const int &r1 = ci;   // ok: both reference and underlying object are const
r1 = 42;              // error: r1 is a reference to const
int &r2 = ci;         // error: non const reference to a const object
```

const的引用

const限定符

- 可以把引用绑定到const对象上，称之为对常量的引用(reference to const)
- 不能通过对常量的引用修改它所绑定的对象

```
const int ci = 1024;
const int &r1 = ci;   // ok: both reference and underlying object are const
r1 = 42;              // error: r1 is a reference to const
int &r2 = ci;         // error: non const reference to a const object
```

初始化

const的引用

- 引用的类型必须与其所引用对象的类型一致，但有两个例外
- 第一种例外情形：初始化常量引用时允许用任意表达式作为初始值，只要该表达式结果能被转换成引用的类型
  - 允许常量引用绑定非常量的对象、字面值，甚至是个一般（右值）表达式

```
int i = 42;
const int &r1 = i;      // we can bind a const int& to a plain int object
const int &r2 = 42;     // ok: r1 is a reference to const
const int &r3 = r1 * 2; // ok: r3 is a reference to const
int &r4 = r1 * 2;       // error: r4 is a plain, non const reference
```

初始化

const的引用

- 当一个常量引用绑定到另外一种类型时，实际上绑定到一个临时对象
  - 临时(temporary)对象是编译器暂存表达式的求值结果时临时创建的一个未命名对象

```
double dval = 3.14;
const int &ri = dval;
```

  - 编译器把上述代码转换为：

```
const int temp = dval;   // create a temporary const int from the double
const int &ri = temp;    // bind ri to that temporary
```

引用非const对象

const的引用

- 常量引用仅对引用可参与的操作做出了限定（只读），但对于引用的对象本身是不是一个常量未作限定

```
int i = 42;
int &r1 = i;          // r1 bound to i
const int &r2 = i;    // r2 also bound to i; but cannot be used to change i
r1 = 0;               // r1 is not const; i is now 0
r2 = 0;               // error: r2 is a reference to const
```

指针和const

- 指向常量的指针(pointer to const)不能用于改变其所指对象的值（只读指针）
  - 要想存放常量对象的地址，只能使用指向常量的指针

```
const double pi = 3.14;   // pi is const; its value may not be changed
double *ptr = &pi;        // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr may point to a double that is const
*cptr = 42;               // error: cannot assign to *cptr
```

指针和const

- 指针的类型必须与其所指对象的类型一致，但有两个例外
- 第一种例外情形：允许一个指向常量的指针指向一个非常量对象
  - 指向常量的指针（只读指针）仅仅要求不能通过该指针改变对象的值

```
double dval = 3.14;      // dval is a double; its value can be changed
const double *cptr = &dval;      // ok: but can't change dval through cptr of ‘const double *’
```

const指针

指针和const

- 常量指针(const pointer)：允许把指针本身定义为常量
  - 把const放在*后用以说明指针是一个常量
- 常量指针必须初始化，之后其值不能再改变（无法再指向其他对象）

```
int errNumb = 0;
int *const curErr = &errNumb;  // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
```

const指针

指针和const

- 能否通过常量指针修改其所指对象的值，取决于所指对象的类型
  - 指向常量的常量指针，不能修改其所指对象的值；而指向非常量的常量指针可以修改

```
const double *const pip = &pi;
*pip = 2.72;     // error: pip is a pointer to const
// if the object to which curErr points (i.e., errNumb) is nonzero
int *const curErr = &errNumb;
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
}
```

练习

◆练习2.27：下面哪些初始化是合法的？

(a) int i = -1, &r = 0;

(b) int *const p2 = &i;

(c) const int i = -1, &r = 0;

(d) const int *const p3 = &i;

(e) const int *p1 = &i;

(f) const int &const r2;

(g) const int i2 = i, &r = i;

练习

指针和const

- 练习2.28：解释如下定义，指出其中不合法的

(a) int i, *const cp;

(b) int *p1, *const p2;

(c) const int ic, &r = ic;

(d) const int *const p3;

(e) const int *p;

- 练习2.29：对上题定义的变量，哪些赋值是合法的？

(a) i = ic;

(b) p1 = p3;

(c) p1 = ⁣

(d) p3 = ⁣

(e) p2 = p1;

(f) ic = *p3;

指针与引用

- 引用本身占用的内存空间中，存储的就是被引用变量的地址
- 引用的功能相当于指针常量
  - 普通指针可以多次被赋值
  - 引用只能在初始化时指定被引用的对象
- 只有常引用，没有引用常量
  - 不能用T & const作为引用类型

指针常量与引用的比较

指针与引用

操作	T类型的指针常量	对T类型的引用
定义并用v初始化	T * const p = &v;	T &r = v;
取v的值	*p	r
访问成员m	p->m	r.m
读取v的地址	p	&r

- p可以再被取地址，而&r则不行
- 引用本身的地址是不可以获得的
- 引用实现的功能，用指针都可以实现

指针与引用的对应关系

指针与引用

```
//使用指针常量
void swap(int * const pa, int *
    const pb) {
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
int main() {
    int a, b;
    ……
    swap(&a, &b);
    ……
    return 0;
}
```

```
//使用引用
void swap(int &ra, int &rb) {
    int temp = ra;
    ra = rb;
    rb = temp;
}
int main() {
    int a, b;
    ……
    swap(a, b);
    ……
    return 0;
}
```

指针与引用的联系

指针与引用

- 引用在底层通过指针来实现
  - 一个引用变量，通过存储被引用对象的地址，来标识它所引用的对象
- 引用是对指针的包装，比指针更高级
  - 指针是C语言就有的底层概念，使用起来很灵活，但用不好容易出错
  - 引用隐藏了指针的“地址”概念，不能直接对地址操作，比指针更安全

引用与指针的选择

指针与引用

- 什么时候用引用？
  - 如无需直接对地址进行操作，指针一般都可用引用代替
  - 用更多的引用代替指针，更简洁、安全
- 什么时候用指针？
  - 引用的功能没有指针强大，有时不得不用指针：
    - ✓引用一经初始化，无法更改被引用对象，如有这种需求，必须用指针；
    - ✓没有空引用，但有空指针，如果空指针有存在的必要，必须用指针；
    - ✓函数指针；
    - ✓用new动态创建的对象或数组，用指针存储其地址最自然；
    - ✓函数调用时，以数组形式传递大量数据时，需要用指针作为参数。

显式类型转换

- 语法形式（3种）：

  类型说明符 (表达式)　　// C++函数风格

  (类型说明符) 表达式　　// C风格

  类型转换操作符 <类型说明符> (表达式)

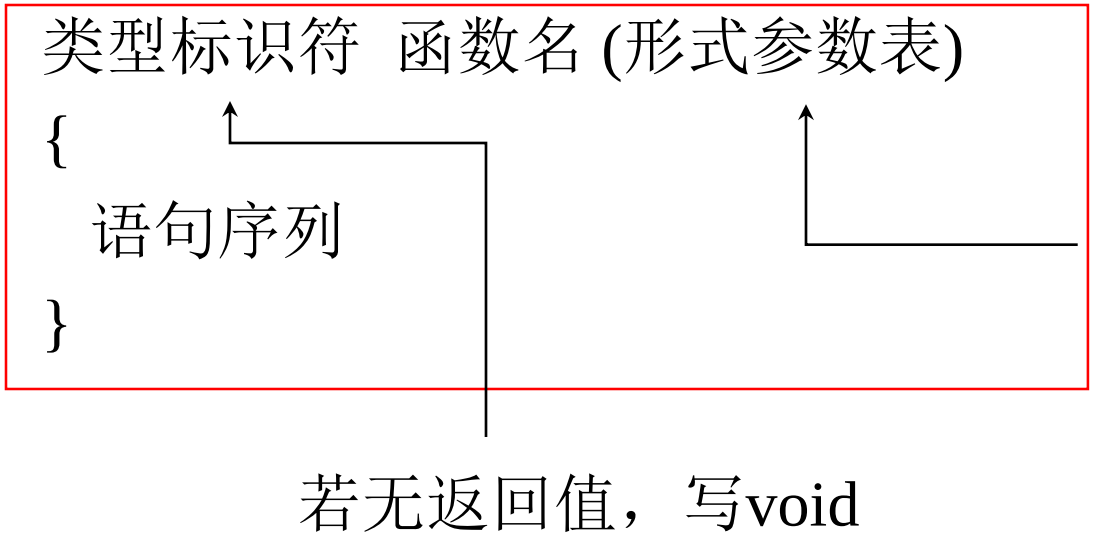
  - 类型转换操作符可以是：
    const_cast、dynamic_cast、
    reinterpret_cast、static_cast
- 显式类型转换的作用是将表达式的结果类型转换为类型说明符所指定的类型。
- 例：int(z), (int)z, static_cast<int>(z)
  三种完全等价

函数

- 函数原型
- 引用形参
- 内联函数 (C99)
- 默认实参
- 函数重载

函数的定义

- 函数是面向对象程序设计中，对功能的抽象
- 函数定义的语法形式



函数的定义

- 形式参数表

  $<type_1>\ name_1, <type_2>\ name_2, ..., <type_n>\ name_n$

- 函数的返回值
  - 由 return 语句给出，例如：
    return 0;
  - 无返回值的函数（void类型），不必写return语句。

函数的调用

- 调用前先声明函数：
  - 若函数定义在调用点之前，则无需另外声明；
  - 若函数定义在调用点之后，则需要在调用函数前按如下形式声明函数原型：

  类型标识符 被调用函数名 (形参列表);
- 调用形式

  函数名 (实参列表)

引用形参

- 传值调用和传引用调用

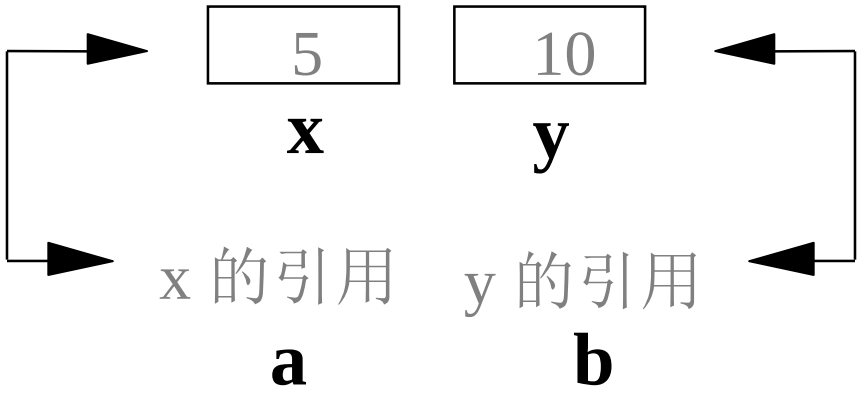
```
#include<iostream>
using namespace std;
void swap(int& a, int& b) {
        int t = a;
        a = b;
        b = t;
}
int main() {
        int x = 5, y = 10;
        cout << "x = " << x << "  y = " << y << endl;
        swap(x, y);
        cout << "x = " << x << "  y = " << y << endl;
        return 0;
}
```

运行结果：

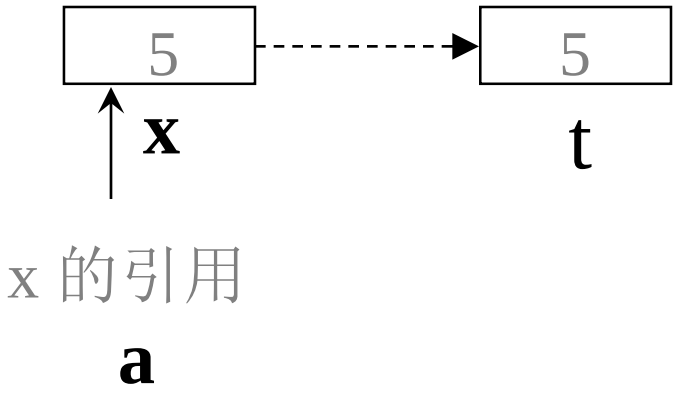
x = 5 y = 10

x = 10 y = 5

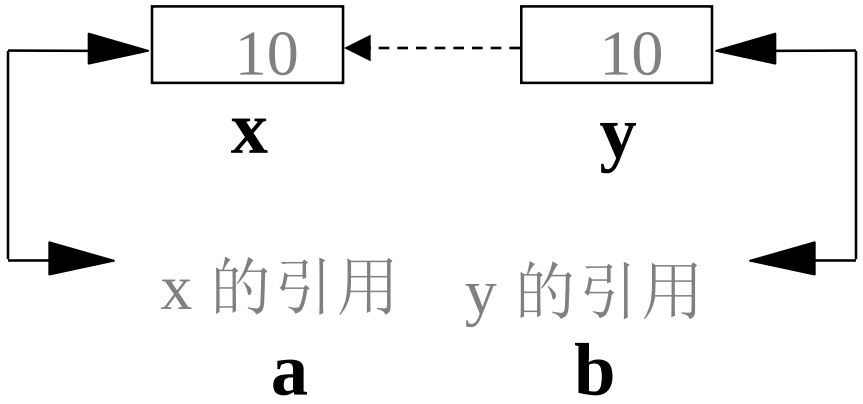
swap(x,y);



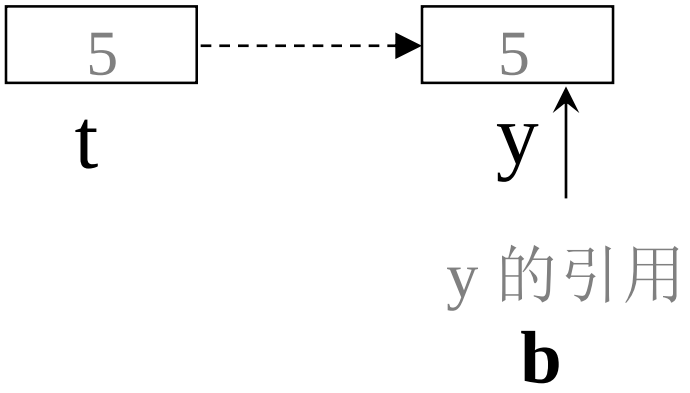
t=a;

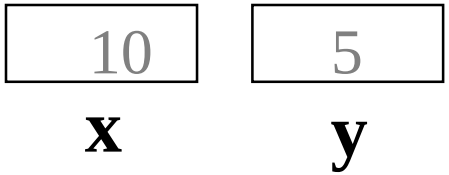


a = b



b = t;





内联函数

- 声明时使用关键字inline。
- 编译时在调用处用函数体进行替换,节省了参数传递、控制转移等开销。
- 注意：
  - 内联函数体内不能有循环语句和switch语句。
  - 内联函数的声明必须出现在内联函数第一次被调用之前。
  - 对内联函数不能进行异常接口声明。

内联函数应用举例


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

const double PI = 3.14159265358979;
inline double calArea(double radius) {
    return PI * radius * radius;
}

int main() {
    double r = 3.0;
    double area          = calArea(r);
    cout << area << endl;
    return 0;
}
```

默认实参

- 函数在声明时可以预先给出默认的实参值，调用时如给出实参，则采用实参值，否则采用预先给出的默认实参值。
- 例如：

```
int add(int x = 5,int y = 6) {
    return x + y;
}
int main() {
    add(10,20); //10+20
    add(10);    //10+6
    add();      //5+6
}
```

默认实参的说明次序

- 有默认实参的形参必须在形参列表的最后，也就是说默认实参值的右面不能有无默认值的参数。因为调用时实参与形参的结合是从左向右的顺序。
- 例：

```
int add(int x, int y = 5, int z = 6);//正确
int add(int x = 1, int y = 5, int z);//错误
int add(int x = 1, int y, int z = 6);//错误
```

默认实参与函数的调用位置

- 如果一个函数有原型声明，且原型声明在定义之前，则默认实参值必须在函数原型声明中给出；而如果只有函数的定义，或函数定义在前，则默认实参值需在函数定义中给出。
- 例：

```
int add(int x = 5,int y = 6);
//原型声明在前
int main() {
  add();
}
int add(int x,int y) {
//此处不能再指定默认值
  return x + y;
}
```

```
int add(int x = 5,int y = 6) {
//只有定义，没有原型声明
  return  x + y;
}
int main() {
  add();
}
```

默认实参的作用域

- 在相同的作用域内，默认实参值的说明应保持唯一；但如果在不同的作用域内，允许说明不同的默认形参。
- 例：

```
int add(int x=1,int y=2);
int main()
{  int add(int x=3,int y=4);
   add();  //使用局部默认形参值（实现3+4）
}
void fun(void)
{   ...
    add();  //使用全局默认形参值（实现1+2）
}
```

练习

默认实参

- 练习：下面的声明是否有错？

  (a) int ff(int a, int b = 0, int c = 0);

  (b) char *init(int ht = 24, int wd, char bckgrnd);

- 练习：下面的调用是否非法？是否有合法但与程序员初衷不符的调用？

  char *init(int ht, int wd = 80, char bckgrnd = ' ');

  (a) init();

  (b) init(24,10);

  (c) init(14, '*');

函数重载

- 函数重载：两个以上的函数，具有相同的函数名，但是形参的个数或者类型不同。

- C中实现整数和浮点数的加法：

```
int iadd(int x, int y);
float fadd(float x, float y);
```

重载函数的声明

- C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载。方便使用，便于记忆。
- 例：

```
int add(int x, int y);
float add(float x, float y);
```

} 形参类型不同

```
int add(int x, int y);
int add(int x, int y, int z);
```

} 形参个数不同

注意事项

- 重载函数的形参必须不同：个数不同或类型不同。
- 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。

```
int add(int x,int y);
int add(int a,int b);
```

编译器不以形参名来区分

```
int add(int x,int y);
void add(int x,int y);
```

编译器不以返回值来区分

- 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆。这样不好：

```
int add(int x, int y)
{  return x + y;  }
```

```
float add(float x,float y)
{  return x - y;  }
```

注意事项

- 当使用具有缺省实参值的函数重载形式时，要防止二义性

```
void fun(int length, int width=2, int height=33);
void fun(int length);

fun(1);
```

- 编译器无法确定执行哪个重载函数，报语法错误

练习

◆练习：说明下面的函数重载声明是否合法

(a) int calc(int, int);

int calc(const int, const int);

(b) int get();

double get();

(c) int *reset(int *);

double *reset(double *);