Chapter1 指针之前

1.基本数据类型和表达式

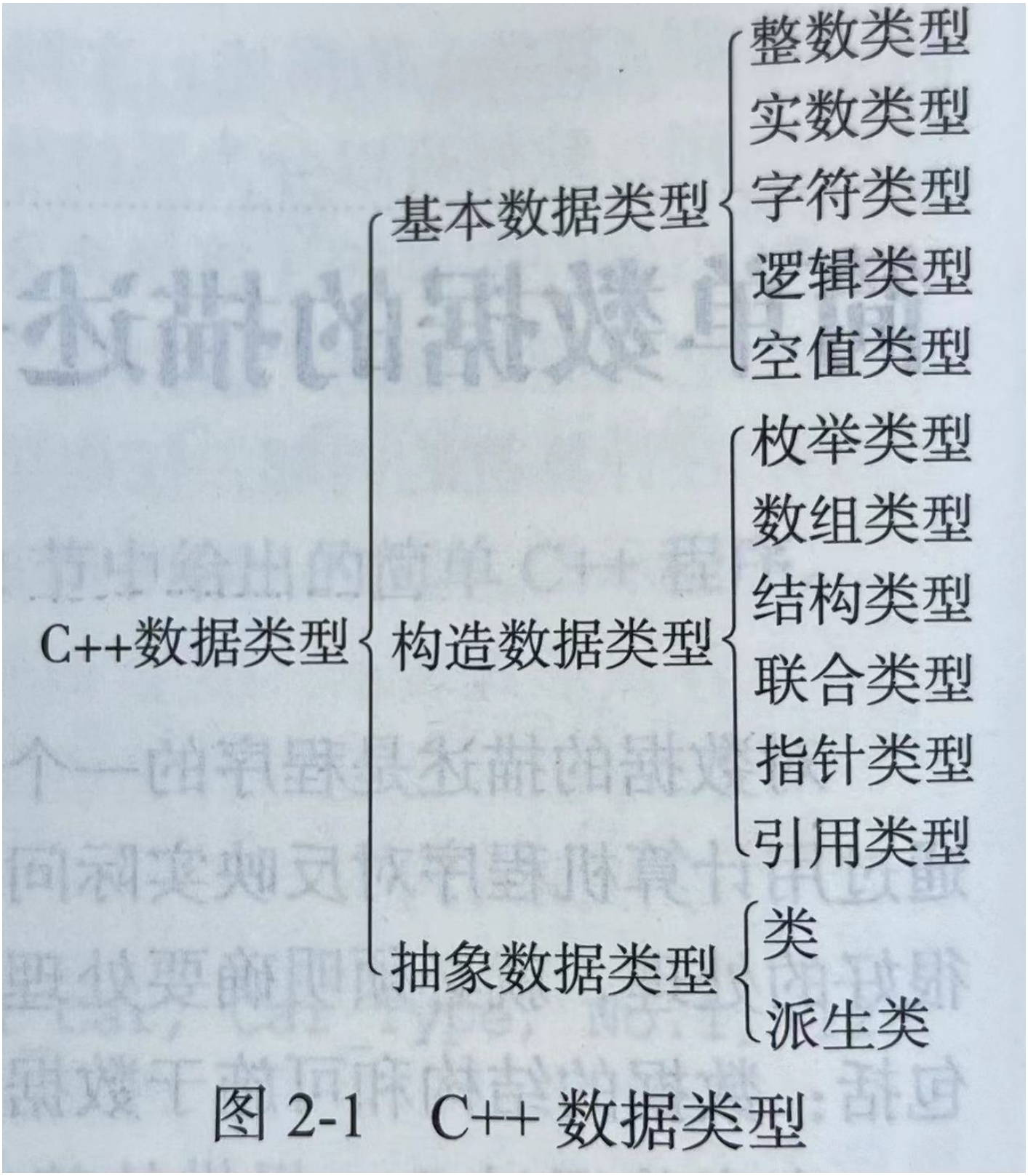


图 2-1 C++ 数据类型

基本数据类型较C增加了：逻辑类型 、 空值类型
构造数据类型没有增加
抽象数据类型增加了：类 和 派生类【支持面向对象】

1.1 基本数据类型

- 整数类型：
  int \ short int或short \long int 或 long
  unsigned int或unsigned \ unsigned short int \ unsigned long int
- 实数类型：
  float \ double \ long double
- 字符类型：
  char 单字节编码字符集中的字符类型数据
  unsigned char \ signed char 在参加算术运算的时候，把字符的编码当做有符号整数还是无符号整数来看待

【补充】
Unicode 国际通用大字符集：包含了大部分语言文字中的字符（英文、中文、日文），2-4字节编码

- 逻辑类型：
  bool
- 空值类型（void）：
  void 没有返回值的函数返回值类型
  void* 通用指针类型

1.2数据表现形式

常量

- 字面常量
  DEF：程序中直接写出常量值的常量
  整数类型字面常量、实数类型字面常量、字符类型、字符串类型字面常量
- 符号常量
  DEF：有名字的常量。在程序中可以通过给常量取一个名字并指定一个类型，然后通过使用这些名字来使用常量

  ```
  const <类型名> <常量名>=<值>;
  或
  #define <常量名>=<值>;
  ```

  ex:
  const double PI=3.14;
  #defien PI=3.14;
  【因使用const，编译程序能够对常量的使用进行更严格的类型检查，所以通常采用第一种常量定义的形式】
  【bool类型也可以看成是C++预定义的两个符号常量，值分别是1和0】

2.程序的流程控制描述-语句

2.1 if

多层if-else是通过
if...
else if
else if

else
来使用的呢

2.2 无条件转移

goto continue

- goto:

```
goto <语句标号>;
<语句标号>:<语句>
```

ex:
goto L1;
L1:
【不能用goto语句从一个函数外部转入到函数的内部，也不能从函数内部转到外部】【不可以跨函数使用goto】

> 可以从内层复合语句->外层复合语句（退出多重循环！！！）。
> 但goto不能跳过带有初始化的变量定义！！！
> 如下图所示：

C++ 允许用 goto 语句从内层复合语句转到外层复合语句或从外层复合语句转入内层复合语句，但是，goto 语句不能掠过带有初始化的变量定义。例如，下面的 goto 语句是非法的：

```
int main()
{  ......
   goto L1;  //Error
   ......
   while (...)
   {  int x=0;
      L1: ...
      ......
      goto L2; //Error
      ......
   }
   ......
   int y=10;
   L2: ......
   ......
}
```

在上述程序中，两个 goto 语句之所以是非法的，是因为它们可能引起标号 L1 和 L2 后面的语句使用未初始化的变量。一般情况下，很少从外层复合语句转入内层复合语句，而从内层复合语句转到外层复合语句有时会用到。例如，可以用 goto 语句退出多重循环（用 break 语句只能退出一层循环）。

- continue:
  DEF：用在循环体语句中，立即结束当前循环，准备进入下一次循环

3.过程抽象-函数

3.1 局部变量和全局变量

- 在定义全局变量之前使用要进行声明：
  `extern <类型名> <变量名>;`（属于非定义声明）
  》》使得编译程序能对变量的操作进行类型检查以及生成高效的可执行代码

- 函数副作用：
  DEF：函数改变了函数调用者的数据（通过在函数中改变非局部变量的值）

3.2 多模块结构

【原则】：按照某种规则对 “全局变量”、“函数及其类型”的定义进行分组，分别放到若干个源文件中
编译程序对每个源文件分别进行编译，编译后通过一个连接程序把“它们各自的编译结果”以及“程序中用到的C++标准库中的代码”连接成一个可执行程序

【模块划分原则】
模块内部内聚性最大（模块内部各个实体之间的管理按程序）
模块之间耦合度最小（各模块之间的一类程序）

> 模块：两个文件-头文件（.h 存储模块接口） 、 源文件（.cpp 存储模块实现）
> 在一个模块中如果要用另外一个模块中定义的全局程序实体，就可以在前者的源文件中用一条编译预处理命令-
> #include，把后者的【头文件】的内容包含进来，达到声明的目的

```
//file.h
extern int x;
extern double y;
int f();

//file1.cpp
int x=1;
double y=2.0;//全局变量x和y
int f()
{
    ...
}

//main.cpp
#include "file1.h"//这样就已经把file1中定义的函数和全局变量，又定义，又声明啦！
int main()
{
    ...
}
```

【补充】
C和其他语言的要求，函数体或复合语句之中，局部数据的定义必须几种写在所有语句的前面！！
如for（i=0;...;...）中不能写int i=0;否则会报错！
C++可以随用随定义！

3.3 变量的生存期

DEF:程序运行时一个变量占有内存空间的时间段称为该变量的生存期

- 静态生存期：从程序开始执行时就进行分配，结束时才收回他们的空间（全局变量）
- 自动生存期：执行到定义他们的复合语句才分配空间。复合语句结束的时候，空间才会被收回（局部变量、函数参数）
- 动态生存期：内存空间使用new或malloc分配，delete或free来收回

定义局部变量的时候，有以下存储类修饰符
1.auto：自动生存期，没有指定存储类的时候，默认是auto

2.static：静态生存期
3.register：建议编译程序把相应的局部变量的空间分配在CPU的寄存器中，提高对局部变量的访问效率

其中，static有两个用处：
① 在全局变量中，static修饰符只让这个变量在本文件中可以使用（文件作用域），别的文件调用此文件时，不可使用这个全局变量
ex：
如果上面file.cpp中定义的是static int x;这样引用file.h的时候，全局变量x也不能被main使用！

② 在局部变量中，static修饰符让这个局部变量变成了这个函数中的全局变量
【且多频繁初始化此变量，只在第一次初始化的时候有效的！】

```
ex:
void f()
{
    extern int x;
    ...//此时可以使用x
    static int x=1;
    ...
}
//此时不能使用x
```

3.4带参数的宏、内联函数、带默认值的形参

由于函数调用时需要开销的（保护调用者的运行环境、指令调用、分配空间...），会带来程序执行效率的下降，特别是一些小函数的频繁使用更加严重啦
为解决上述问题：提供了“带参数的宏”以及“内联函数”来解决

3.4.1 带参数的宏

```
#define <宏名> (<参数表>) <文字串>
```

把出现“宏名”的地方用“文字串”进行替换
在“文字串”中出现的，由“参数表”列出的参数（形参），将被替换成使用该“宏名”的地方所提供的的参数（实参）

ex：
`#define max(a,b) (((a)>(b))?(a):(b))`
此时必须一个“变量”有一个括号，以防a和b是一个表达式，引起优先级错乱的问题呢

3.4.2 内联函数

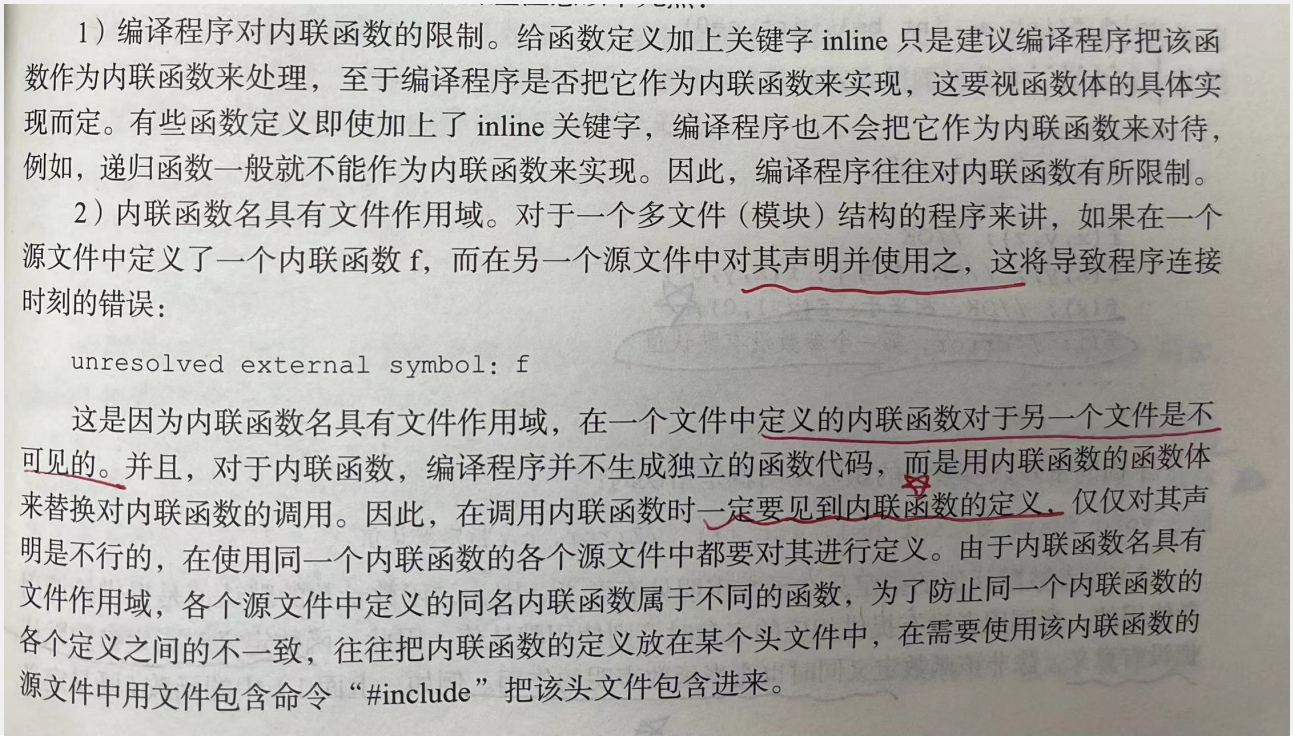
DEF:函数定义中的返回值类型之前加上一个关键字incline
》》【建议编译程序把该函数的函数体展开到调用点】
》》【避免函数调用的开销，提高函数调用的效率】

```
incline int max(int a,int b)
{
    return a>b?a:b;
}
```

> 优点：
> 【形式上属于函数，效果上具有宏定义的高效率！】

缺点：

- 编译程序对内联函数的限制
- 具有文件作用域！



1）编译程序对内联函数的限制。给函数定义加上关键字 inline 只是建议编译程序把该函数作为内联函数来处理，至于编译程序是否把它作为内联函数来实现，这要视函数体的具体实现而定。有些函数定义即使加上了 inline 关键字，编译程序也不会把它作为内联函数来对待，例如，递归函数一般就不能作为内联函数来实现。因此，编译程序往往对内联函数有所限制。

2）内联函数名具有文件作用域。对于一个多文件（模块）结构的程序来讲，如果在一个源文件中定义了一个内联函数 f，而在另一个源文件中对其声明并使用之，这将导致程序连接时刻的错误：

```
unresolved external symbol: f
```

这是因为内联函数名具有文件作用域，在一个文件中定义的内联函数对于另一个文件是不可见的。并且，对于内联函数，编译程序并不生成独立的函数代码，而是用内联函数的函数体来替换对内联函数的调用。因此，在调用内联函数时一定要见到内联函数的定义，仅仅对其声明是不行的，在使用同一个内联函数的各个源文件中都要对其进行定义。由于内联函数名具有文件作用域，各个源文件中定义的同名内联函数属于不同的函数，为了防止同一个内联函数的各个定义之间的不一致，往往把内联函数的定义放在某个头文件中，在需要使用该内联函数的源文件中用文件包含命令“#include”把该头文件包含进来。

3.4.3 带默认值的形参

DEF:在定义或声明函数的时，为函数的某些参数指定默认值。如果调用这些带默认形参的函数时没有提供相应的实参，则相应的形参采用指定的默认值。否则，采用调用者提供的实参

```
声明时：
void print(int value,int base=10);
定义时（除非在调用之前定义，同时充当声明，否则默认值不算数！！！）：
void print(int value,int base=10)
{
    ...
}
```

- 有默认值的应该处于形参表的最右侧！！
- 【对参数默认值的指定只在函数声明处有意义。函数定义时指定参数默认值没有意义！
  除非在调用之前定义，同时充当声明，否则默认值不算数！！！】
- 不同文件，可以对同一个函数，有不同默认值的声明！！！

```
ex:
//file.cpp
void f(int a,int b)
{
    ...
}

//file1.cpp
#include "file.cpp"
void f(int a,int b=0);//声明

//file2.cpp
#include "file.cpp"
void f(int a=3,int b=1);//声明
```

3.5 函数名重载、lambda表达式（匿名函数）

对于一些功能相同，参数类型或个数不同的函数，有时给他们取相同的名字会带来使用上的方便!

下面的 4 个输出函数，它们将分别输出 int 型、double 型、char 型以及自定义类型 A 的数据：

```
void print_int(int i) { ..... }
void print_double(double d) { ..... }
void print_char(char c) { ..... }
void print_A(A a) { ..... } //A为自定义类型
```

在使用上述函数进行输出时，不但要记住这些函数的不同名字，而且要根据待输出数据的类型来选择相应的函数，这将给使用带来麻烦。如果给这些函数取相同的名字，将会减少使用上的一些不便，例如，我们可以把上述的函数定义成：

```
void print(int i) { ..... }
void print(double d) { ..... }
void print(char c) { ..... }
void print(A a) { ..... }
```

在使用这些函数时，由调用者提供的实参的类型来决定实际调用的是哪一个函数。例如，print(1.0) 将调用上面的第二个函数：void print(double d)。

DEF：具有相同的功能，但参数和函数体的实现有所不同。给多个函数取相同名字的语言机制称为函数名重载

3.5.1 重载函数的定义

原则：参数的类型或个数有所不同

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

3.5.2 重载函数调用的绑定

绑定DEF：定义的多个重名函数，确定一个对重载函数的调用对应着哪一个重载函数定义的过程称为“绑定”

匹配：

构造一个候选集，此候选集由形参个数与实参相同的重载函数构成，按预定的类型匹配规则从该候选集中选择最佳匹配

（1）精确匹配

精确匹配是指实参与某个重载函数的形参类型完全相同或通过一些“细微”的转换之后相同。这里，细微的转换（trivial conversion）包括：数组名可转换成其第一个元素的指针（将在5.5.5节中介绍）和函数名可转换成函数指针（将在5.5.6节中介绍）等。例如，对于下述的重载函数：

```
void print(int);
void print(double);
void print(char);
```

下面的函数调用可根据精确匹配进行绑定：

```
print(1); //绑定到函数: void print(int);
print(1.0); //绑定到函数: void print(double);
print('a'); //绑定到函数: void print(char);
```

（2）提升匹配

如果精确匹配不成功，则进一步尝试提升匹配。它首先对实参进行提升转换，然后，用转换后的实参与重载函数的形参进行精确匹配。提升转换规则如下：

1）按整型提升规则（参见2.4.7节）提升。

2）把float提升到double或把double提升到long double。

例如，对于下述的重载函数：

```
void print(int);
void print(double);
```

根据提升匹配，下面的函数调用：

```
print('a'); //绑定到函数: void print(int);
print(1.0f); //绑定到函数: void print(double);
```

（3）标准转换匹配

如果精确匹配和提升匹配都不成功，则再尝试标准转换匹配。它首先对实参进行标准转换，然后，用转换后的实参与重载函数的形参进行精确匹配。标准转换（standard conversion）规则主要包括：

1）任何算术型之间可以互相转换。

2）枚举类型可以转换成任何算术型。

3）零可以转换成任何算术型或指针类型。

4）任何类型的指针可以转换成void *。

5）派生类指针可以转换成基类指针。

6）每个标准转换都是平等的。（这一条表明：1）～5）的优先级相同。）

例如，对于下面的重载函数：

```
void print(char);
void print(char *);
```

根据标准转换匹配，下面的函数调用将绑定到函数void print(char)：

```
print(1); //绑定到函数: void print(char);
```

（4）自定义转换匹配

如果精确匹配、提升匹配以及标准转换匹配都不成功，则进行自定义转换匹配。它首先对实参实施自定义类型转换（自定义类型转换将在 6.6.5 节中介绍），然后，用转换后的实参与重载函数的形参进行精确匹配、提升匹配或标准转换匹配。

- 无法匹配或匹配不唯一》》绑定失败
- 使用带默认值的函数声明可以减少函数重载的数量！！！

3.5.3 lambda表达式

`[<环境变量使用说明>](<形式参数>)-><返回值类型指定><函数体>`

ex:

```
int n=[](int x,int y)->int{return x*y;}(3,4);
```

【环境变量使用说明】：指出函数体中对外层作用域中的自动变量的使用限制

- 空：不能使用外层作用域中的自动变量
- &：按引用方式使用外层作用域中的自动变量（可以改变他们的值）
- =：按值方式使用外层作用域中的自动变量（不能改变他们的值）

下面是一些合法的 λ 表达式：

```
{  int k,m,n;//环境变量
   ...
   ...[](int x)->int { return x; }...//不能使用k、m、n
   ...[&](int x)->int { k++; m++; n++; return x+k+m+n; }...//k、m、n可
                                                          //以被修改
   ...[=](int x)->int { return x+k+m+n; }...//k、m、n不能被修改
   ...[&,n](int x)->int { k++; m++; return x+k+m+n; }...//n不能被修改
   ...[=,&n](int x)->int { n++; return x+k+m+n; }...//n可以被修改
   ...[&k,m](int x)->int { k++; return x+k+m; }...//只能使用k和m，k可以
                                                  //被修改
   ...[=] { return k+m+n; }...//没有参数，返回值类型为int（编译器自动确定）
}
```

在 C++ 中，λ 表达式通常用于把一个匿名函数作为参数传给另一个函数的场合（详细内容将在 5.5.6 节中介绍），而 λ 表达式则是通过函数对象来实现的（详细内容将在 6.6.5 节中介绍）。