第二章　基本数据类型和表达式

目录

学习目标

1. 理解基本数据类型的内存结构；
2. 理解对象的基本属性；
3. 学会运用 const 修饰符和类型自动推导；
4. 掌握表达式求值的基本方法。

2.1 C++ 语句基本元素

C++ 字符集

包括大小写字母、阿拉伯数字以及符号：

```
+  -  *  /  =  ,  .  _  :  ;  ?  "  '  ~  |  !  #
%  &  ()  []  {}  ^  <  > 和空格等
```

2.1 C++ 语句基本元素

C++ 字符集

包括大小写字母、阿拉伯数字以及符号：

```
+   -   *   /   =   ,   .   _   :   ;   ?   "   '   ~   |   !   #
%   &   ()   []   {}   ^   <   > 和空格等
```

标识符

- 是由用户定义的“单词”，用来给变量、常量、数据类型、函数等命名
- 合法的标识符由字母、数字和下划线组成，且必须以字母或下划线开头
- 严格区分大小写，如:name、Name 和 NAME 是三个不同的标识符

2.1 C++ 语句基本元素

标识符命名小建议

- 对象名一般小写，如 name, 而不是 NAME;
- 应使用能帮助记忆的名字，如 salary，而不是 s;
- 有多个单词组成时，单词之间可用下划线或是内嵌单词的第一个字母大写;

  如 student_name 或 studentName。

2.1 C++ 语句基本元素

关键字

- C++ 语言定义的一些供自身使用、且有特殊含义的英文单词
- 关键字不能用作用户自定义标识符
- 关键字全部由小写字母组成

2.1 C++ 语句基本元素

C++ 关键字

alignas§	alignof§	asm	auto†	bool
break	case	catch	char16_t§	char
char32_t§	class†	const	constexpr§	const_cast
continue	decltype§	default†	delete†	do
double	dynamic_cast	else	enum	explicit
export†	extern†	false	float	for
friend	if	goto	inline†	int
long	mutable†	namespace	noexcept§	new
nullptr§	operator	private	protected	public
register‡	reinterpret_cast	return	short	signed
sizeof†	static	static_cast	static_assert§	struct†
switch	template	this	thread_local§	throw
true	try	typedef	typeid	typename
union	unsigned	using†	virtual	void
volatile	wchar_t	while		

§ C++11 标准新增关键字。
† 在 C++11 标准下含义发生了变化或者增加了新含义。
‡ 在 C++17 标准下含义发生了变化。

2.1 C++ 语句基本元素

例：判断下面哪些是合法的用户标示符?

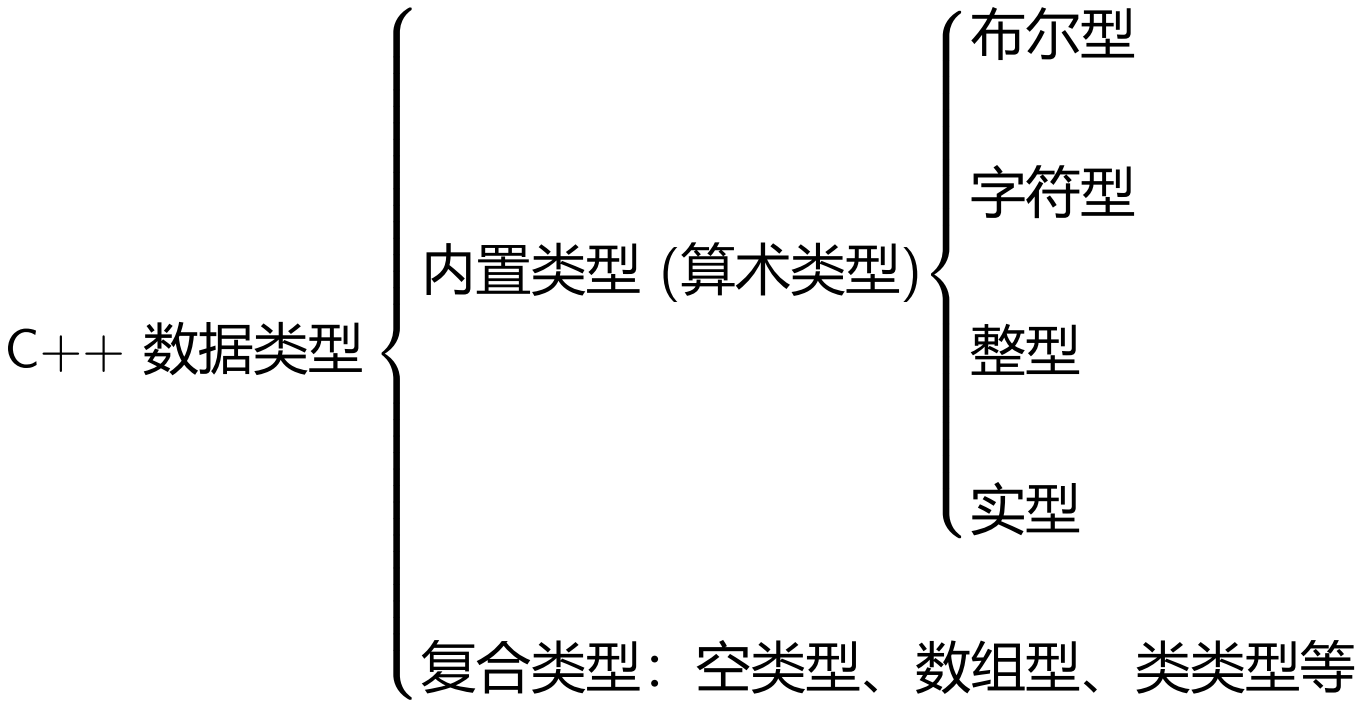
MyFile	94Salary
Salary 94	Salary94
amount	$amount
void	f3.5
Num_of_Student	name_5

2.1 C++ 语句基本元素

例：判断下面哪些是合法的用户表示符?

MyFile	~~94Salary~~
~~Salary 94~~	Salary94
amount	~~$amount~~
~~void~~	~~f3.5~~
Num_of_Student	name_5

2.2 基本数据类型



2.2 基本数据类型 — 内置类型

C++ 基本内置类型

类型	含义	尺寸（单位: 字节 (byte)）	
		最小尺寸/精度	Visual C++
bool	布尔类型	未定义	1
char	字符类型	1	1
wchar_t	宽字符	2	2
char16_t	Unicode 字符	2	2
char32_t	Unicode 字符	4	4
short	短整型	2	2
int	整型	2	4
long	长整型	4	4
long long	双长整型	8	8
float	单精度浮点数	6 位有效数字（IEEE 754）	4
double	双精度浮点数	15 位有效数字（IEEE 754）	8
long double	扩展的精度浮点数	精度不低于 double 类型	8

位（bit）是计算机中存储数据的最小单位，指二进制数中的一个位数，其值为 0 或 1。
字节（byte）是计算机存储容量的基本单位，一个字节由 8 位二进制数组成，即 8 个比特位。

2.2 基本数据类型 — 内置类型

布尔型（bool）

- 取值为 `true`(真) 或 `false`(假)。

2.2 基本数据类型 — 内置类型

布尔型（bool）

- 取值为 true(真) 或 false(假)。

字符型（char）

- 常用来保存字符，存储的是该字符的 ASCII 码值，占一个字节，如：'A'，'a' 的 ASCII 码值分别为 65 和 97
- C++ 支持扩展的字符集，如 wchar_t、char16_t 和 char32_t。wchar_t 又称双字节字符类型，可以存放扩展字符集中任意一个字符，比如中文字符

2.2 基本数据类型 — 内置类型

整型（int）

- 用来保存整数，可分为带符号的（signed）和无符号的（unsigned）两种，如-1，1。
- 二进制整数表示
  原码，3：0000 0011，-3：1000 0011
  反码，3：0000 0011，-3：1111 1100
  补码，3：0000 0011，-3：1111 1101

2.2 基本数据类型 — 内置类型

整型（int）

- 用来保存整数，可分为带符号的（signed）和无符号的（unsigned）两种，如-1，1。
- 二进制整数表示
  原码，3：0000 0011，-3：1000 0011
  反码，3：0000 0011，-3：1111 1100
  补码，3：0000 0011，-3：1111 1101

思考：一个 char 类型表示范围？

2.2 基本数据类型 — 内置类型

整型（int）

- 用来保存整数，可分为带符号的（signed）和无符号的（unsigned）两种，如-1，1。
- 二进制整数表示
  原码，3：0000 0011，-3：1000 0011
  反码，3：0000 0011，-3：1111 1100
  补码，3：0000 0011，-3：1111 1101

思考：一个 char 类型表示范围?

一个 char 类型表示范围为 [1000 0000, 0111 1111] 即-128 至 127。

2.2 基本数据类型 — 内置类型

整型（int）

- 用来保存整数，可分为带符号的（signed）和无符号的（unsigned）两种，如-1，1。
- 二进制整数表示
  原码，3：0000 0011，-3：1000 0011
  反码，3：0000 0011，-3：1111 1100
  补码，3：0000 0011，-3：1111 1101

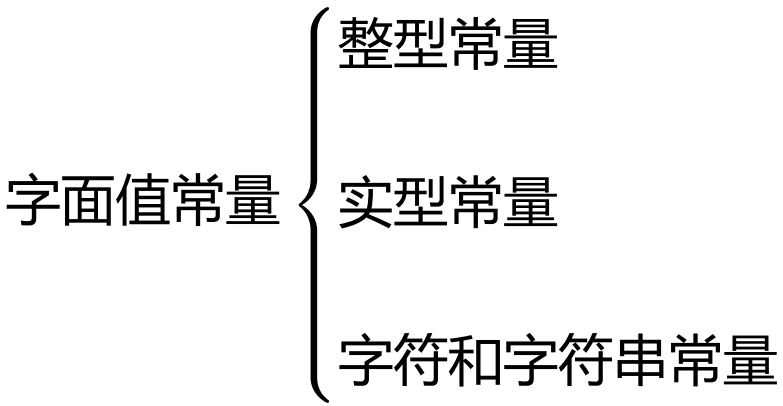
思考：一个 char 类型表示范围?

一个 char 类型表示范围为 [1000 0000, 0111 1111] 即-128 至 127。

浮点型

- 可表示单精度（float，4B）、双精度（double，8B）和扩展精度（long double，8B、12B 或 16B），用来保存实数，如 3.1415926。

2.2 基本数据类型 — 常量（字面值常量）



2.2 基本数据类型 — 常量（字面值常量）

整型常量的表示方法

- 十进制整数：如 58

2.2 基本数据类型 — 常量（字面值常量）

整型常量的表示方法

- 十进制整数：如 58
- 八进制整数（以0开头，0-7组成）：如 072

2.2 基本数据类型 — 常量（字面值常量）

整型常量的表示方法

- 十进制整数：如 58
- 八进制整数（以0开头，0-7组成）：如 072
- 十六进制整数（以0x或0X开头，由数字0-9和字母A-F(大小写均可)）：如 0x3A

2.2 基本数据类型 — 常量（字面值常量）

整型常量的表示方法

- 十进制整数：如 58
- 八进制整数（以0开头，0-7组成）：如 072
- 十六进制整数（以0x或0X开头，由数字0-9和字母A-F(大小写均可)）：如 0x3A

实型常量的表示方法

- 小数形式：如 3.14159　　4.235
- 指数形式：如 3.14159E0　　4e6

说明

指数形式由尾数、阶数和 E 或 e组成，其中在 E 或 e 前面的尾数部分必须有数字，后面的阶数必须为整数

2.2 基本数据类型 — 常量（字面值常量）

字符和字符串常量

- 字符常量：由单引号括起来单个字符，如：`'a'` `'1'`
- 字符串常量：由双引号括起来的字符，如：`"a"` `"Mandy"`。自动在末尾添加空字符（`'\0'`）
- 特殊字符：用转义序列表示方法，如：\n 换行符、\r 回车符，\\ 反斜线等
  \nnn 八进制数，\xnn 十六进制数，如 `'\141'` 和`'\x61'` 都表示字符`'a'`

```
cout << "Hi␣\"\103++\"\n"; // 输出Hi "C++", 转到新一行
```

2.2 基本数据类型 — 常量（字面值常量）

字符和字符串常量

- 字符常量：由单引号括起来单个字符，如：`'a'` `'1'`
- 字符串常量：由双引号括起来的字符，如：`"a"` `"Mandy"`。自动在末尾添加空字符（`'\0'`）
- 特殊字符：用转义序列表示方法，如：\n 换行符、\r 回车符，\\ 反斜线等
  \nnn 八进制数，\xnn 十六进制数，如 `'\141'` 和`'\x61'` 都表示字符`'a'`

```
cout << "Hi␣\"\103++\"\n"; // 输出Hi "C++", 转到新一行
```

前缀和后缀

- 为整型、实型和字符型常量添加前缀或后缀，可以改变其默认类型
- 例如：

`3.14159L`	扩展精度实型字面常量，类型为 `long double`
`-84L`	长整型，类型为 `long int`
`3.14E-3F`	单精度实型常量，类型为 `float`
`0x2aLU`	十六进制表示的无符号长整型数 42，类型为 `unsigned long int`

2.3 对象 — 对象的定义与初始化

对象

- 对象是数据和操作的载体，当创建一个对象时，会为它分配一定内存空间
- 具有数据类型、名字、内存结构、支持的操作、生命期和作用域等属性

2.3 对象 — 对象的定义与初始化

对象

- 对象是数据和操作的载体，当创建一个对象时，会为它分配一定内存空间
- 具有数据类型、名字、内存结构、支持的操作、生命期和作用域等属性

对象的定义

一般格式：< 数据类型名 > 对象名 ;

例如：

```
int counter;      //定义一个整型类型对象
Cylinder object; //定义一个Cylinder类类型对象
```

2.3 对象 — 对象的定义与初始化

对象的初始化

```
int age = 10;       //复制初始化
int year(2017);  //直接初始化
int age1 = 20, age2 = age1; //用先定义的对象的值初始化后面定义的对象
```

C++11 新标准

```
int year{2017};  //列表初始化
int year = {2017}; //列表初始化
int year{};       //可以不提供初始值，默认为0
```

初始化与赋值

```
int year = 0; //定义对象并初始化，在此之前， year在内存中不存在
year = 2017; //赋值操作，在对 year赋值之前，其在内存中已经存在
```

2.3 对象 — 对象的声明

为了支持在程序文件之间共享代码，C++ 引入了声明机制

2.3 对象 — 对象的声明

为了支持在程序文件之间共享代码，C++ 引入了声明机制

声明对象需要利用关键字extern

例如

```
int i(1);      //在一个源文件里面定义对象 i 并初始化
extern int i; //在另外一个文件里面声明对象i，i已经定义过了
```

2.3 对象 — 对象的声明

为了支持在程序文件之间共享代码，C++ 引入了声明机制

声明对象需要利用关键字extern

例如

```
int i(1);     //在一个源文件里面定义对象 i 并初始化
extern int i; //在另外一个文件里面声明对象i，i已经定义过了
```

利用 extern 声明对象时提供一个初始值，这时声明就变成了定义

例如

```
extern int i = 0; //定义i
int i = 5;        //错误：对象i已经定义过了
```

2.3 对象 — 对象的声明

为了支持在程序文件之间共享代码，C++ 引入了声明机制

声明对象需要利用关键字extern

例如

```
int i(1);     //在一个源文件里面定义对象 i 并初始化
extern int i; //在另外一个文件里面声明对象i，i已经定义过了
```

提问

声明和定义有何区别?

利用 extern 声明对象时提供一个初始值，这时声明就变成了定义

例如

```
extern int i = 0; //定义i
int i = 5;        //错误：对象i已经定义过了
```

2.3 对象 — 对象的声明

为了支持在程序文件之间共享代码，C++ 引入了声明机制

声明对象需要利用关键字extern

例如

```
int i(1);     //在一个源文件里面定义对象 i 并初始化
extern int i; //在另外一个文件里面声明对象i，i已经定义过了
```

提问

声明和定义有何区别?

利用 extern 声明对象时提供一个初始值，这时声明就变成了定义

例如

```
extern int i = 0; //定义i
int i = 5;        //错误：对象i已经定义过了
```

答案

一个对象只能定义一次，但可以声明多次。如左边的第二行代码因重复定义 i 而出错

2.3 对象 — 作用域和生命期

作用域：指标识符在代码中的可见范围，通常以花括号分隔
生命期：具有块域的对象，通常其生命期从定义处开始，在作用域结束时消亡

2.3 对象 — 作用域和生命期

作用域：指标识符在代码中的可见范围，通常以花括号分隔
生命期：具有块域的对象，通常其生命期从定义处开始，在作用域结束时消亡

示例

```
1  #include<iostream>
2  using namespace std;
3  int main() {
4      int sum = 0; //存放两个数的和
5      {
6          int val1 = 10, val2 = 10;
7          sum = val1 + val2;
8      }
9      cout << sum;
10     return 0;
11 }
```

说明

- sum 的作用域从第 4 行开始到函数体的结束处 (第 11 行) 结束
- val1 和 val2 的作用域从第 6 行开始，到第 8 行结束

2.3 对象 — 作用域和生命期

作用域嵌套：如果一个块域包含了另外一个块域，这就构成了作用域的嵌套

示例

```
1  #include<iostream>
2  int main() {
3      int sum = 10;
4      {
5          int sum = 0;
6          //访问内层sum，打印输出0
7          std::cout << sum << '\n';
8      }
9      //访问外层sum，打印输出10
10     std::cout << sum << std::endl;
11     return 0;
12 }
```

2.3 对象 — 作用域和生命期

作用域嵌套：如果一个块域包含了另外一个块域，这就构成了作用域的嵌套

示例

```
1  #include<iostream>
2  int main() {
3      int sum = 10;
4      {
5          int sum = 0;
6          //访问内层sum，打印输出0
7          std::cout << sum << '\n';
8      }
9      //访问外层sum，打印输出10
10     std::cout << sum << std::endl;
11     return 0;
12 }
```

说明

C++ 采用局部优先的原则，即外层对象的作用域被内层同名对象的作用域屏蔽.

- 第 7 行访问的是内层对象 sum
- 第 10 行访问的是外层对象 sum

2.3 对象 — 作用域和生命期

作用域嵌套：如果一个块域包含了另外一个块域，这就构成了作用域的嵌套

示例

```
1  #include<iostream>
2  int main() {
3      int sum = 10;
4      {
5          int sum = 0;
6          //访问内层sum，打印输出0
7          std::cout << sum << '\n';
8      }
9      //访问外层sum，打印输出10
10     std::cout << sum << std::endl;
11     return 0;
12 }
```

说明

C++ 采用局部优先的原则，即外层对象的作用域被内层同名对象的作用域屏蔽.

- 第 7 行访问的是内层对象 sum
- 第 10 行访问的是外层对象 sum

建议

- 对象在使用的时候定义
- 内层对象的命名不要和外层对象的名字相同

2.4 常量修饰符和类型推导 — const 修饰符

有时，对于存储圆周率 π、自然对数 e 等常量的对象，我们不希望其内容发生变化。C++ 提供了关键字const对对象的类型加以限制。

例如

```
//圆周率用pi表示，即有时给常量取个名字更方便
const double pi = 3.14159;
int i = 100;
//利用对象 i 的值初始化 ci
const int ci = i;
```

2.4 常量修饰符和类型推导 — const 修饰符

有时，对于存储圆周率 π、自然对数 e 等常量的对象，我们不希望其内容发生变化。C++ 提供了关键字const对对象的类型加以限制。

例如

```
//圆周率用pi表示，即有时给常量取个名字更方便
const double pi = 3.14159;
int i = 100;
//利用对象 i 的值初始化 ci
const int ci = i;
```

说明

- const 修饰的对象不能改变其内容，即不能对其进行写操作
- const 对象创建时必须初始化

2.4 常量修饰符和类型推导 — const 修饰符

有时，对于存储圆周率 π、自然对数 e 等常量的对象，我们不希望其内容发生变化。C++ 提供了关键字const对对象的类型加以限制。

例如

```
//圆周率用pi表示，即有时给常量取个名字更方便
const double pi = 3.14159;
int i = 100;
//利用对象 i 的值初始化 ci
const int ci = i;
```

说明

- const 修饰的对象不能改变其内容，即不能对其进行写操作
- const 对象创建时必须初始化

问题

判断如下代码是否正确?

```
const int numStudent = 30;
numStudent = 50;
const double pi;
```

2.4 常量修饰符和类型推导 — const 修饰符

有时，对于存储圆周率 π、自然对数 e 等常量的对象，我们不希望其内容发生变化。C++ 提供了关键字const对对象的类型加以限制。

例如

```
//圆周率用pi表示，即有时给常量取个名字更方便
const double pi = 3.14159;
int i = 100;
//利用对象 i 的值初始化 ci
const int ci = i;
```

说明

- const 修饰的对象不能改变其内容，即不能对其进行写操作
- const 对象创建时必须初始化

问题

判断如下代码是否正确?

```
const int numStudent = 30;
numStudent = 50;
const double pi;
```

答案

- 第二行代码错误：不能对 numStudent 进行写值操作
- 第三行代码错误：const 对象必须初始化

2.4 常量修饰符和类型推导 — constexpr 和常量表达式

对比如下代码一和代码二的常量定义有何区别?

代码一

```
const int a = 3;
const int b = 2 + 5;
```

代码二

```
int num = 100;
const int numMax = num;
```

2.4 常量修饰符和类型推导 — constexpr 和常量表达式

对比如下代码一和代码二的常量定义有何区别?

代码一

```
const int a = 3;
const int b = 2 + 5;
```

代码二

```
int num = 100;
const int numMax = num;
```

答案

- 代码一中的 const 对象 a，b 分别是用字面值常量、和常量表达式初始化的
- 代码二中的 const 对象 numMax 是用对象 num 初始化的

2.4 常量修饰符和类型推导 — constexpr 和常量表达式

对比如下代码一和代码二的常量定义有何区别?

代码一

```
const int a = 3;
const int b = 2 + 5;
```

代码二

```
int num = 100;
const int numMax = num;
```

答案

- 代码一中的 const 对象 a，b 分别是用字面值常量、和常量表达式初始化的
- 代码二中的 const 对象 numMax 是用对象 num 初始化的

说明

- 常量表达式的值不会改变，且在编译期间就能得到计算结果
- 代码一中的 a 和 b 均为编译时常量(用常量表达式初始化的常量)
- 代码二中的 numMax 的值只有在程序运行期间可以获取，即为运行时常量

2.4 常量修饰符和类型推导 — constexpr 和常量表达式

为了帮助编译器自动识别常量表达式，C++11 提供了 constexpr 关键字

例如

```
constexpr int number = 10;          // 10 是常量表达式
constexpr int maxNumber = number + 1; //number+1 是常量表达式
```

2.4 常量修饰符和类型推导 — constexpr 和常量表达式

为了帮助编译器自动识别常量表达式，C++11 提供了 constexpr 关键字

例如

```
constexpr int number = 10;          // 10 是常量表达式
constexpr int maxNumber = number + 1; //number+1 是常量表达式
```

注意

constexpr 修饰的对象是一个常量，而且必须用常量表达式初始化

2.4 常量修饰符和类型推导 — 类型推导

为了使代码更易于理解和使用，有时把已经定义的数据类型换个新的名字，即类型别名。有两种方法定义类型别名，关键字 `typedef`和`using`。

2.4 常量修饰符和类型推导 — 类型推导

为了使代码更易于理解和使用，有时把已经定义的数据类型换个新的名字，即类型别名。有两种方法定义类型别名，关键字 typedef和using。

例如

方法一：

```
// price是 double的一个类型别名
typedef double price;
//car和 mobile存放的都是价格
price car=1.0e5, mobile=100.;
```

方法二：

```
using price = double;
```

2.4 常量修饰符和类型推导 — 类型推导

为了使代码更易于理解和使用，有时把已经定义的数据类型换个新的名字，即类型别名。有两种方法定义类型别名，关键字 typedef和using。

例如

方法一：

```
// price是 double的一个类型别名
typedef double price;
//car和 mobile存放的都是价格
price car=1.0e5, mobile=100.;
```

方法二：

```
using price = double;
```

建议

由于 using 声明更符合我们的编程习惯，其使用方式和定义对象的方式类似，故推荐使用 using 声明

2.4 常量修饰符和类型推导 — 类型推导

```
int i = 0;
```

对于上面这条代码，能否根据操作数 0 的类型（int）自动推断出 i 的类型?

2.4 常量修饰符和类型推导 — 类型推导

```
int i = 0;
```

对于上面这条代码，能否根据操作数 0 的类型（int）自动推断出 i 的类型?

答

利用 auto，编译器可以根据初始值的类型自动推导出所需数据类型

```
auto pi=3.14159, rad=1.0; // pi 和 rad 都为 double 类型
auto area=pi*rad*rad;   // area 为 double 类型
auto i=0, pi=3.14159;   // 错误: i 和 pi 的类型不一致
```

2.4 常量修饰符和类型推导 — 类型推导

```
int i = 0;
```

对于上面这条代码，能否根据操作数 0 的类型（int）自动推断出 i 的类型?

答

利用 auto，编译器可以根据初始值的类型自动推导出所需数据类型

```
auto pi=3.14159, rad=1.0; // pi 和 rad 都为 double 类型
auto area=pi*rad*rad;   // area 为 double 类型
auto i=0, pi=3.14159;   // 错误: i 和 pi 的类型不一致
```

注意

- 当用 auto 定义多个对象时，其显示类型必须一致，否则会报错
- auto 不能肆意使用，否则会造成代码的可读性和可维护性下降，使用时需要权衡利弊

2.4 常量修饰符和类型推导 — 类型推导

有时只想用表达式的类型来定义对象，为此 C++11 引入了 decltype 关键字，它能够在不用计算表达式的情况下获取表达式的数据类型

2.4 常量修饰符和类型推导 — 类型推导

有时只想用表达式的类型来定义对象，为此 C++11 引入了 decltype 关键字，它能够在不用计算表达式的情况下获取表达式的数据类型

例如

```
int i = 0;
decltype (i) j = 1; //j 为 int 类型
decltype (i + j) k = 0; //k 为 int 类型
```

说明

decltype 分析 i + j 的值的数据类型，但不会计算 i + j 的值。

2.4 常量修饰符和类型推导 — 类型推导

当 `auto` 和 `decltype` 遇到 const 会怎样呢?

2.4 常量修饰符和类型推导 — 类型推导

当 auto 和 decltype 遇到 const 会怎样呢?

auto

```
auto将忽略掉 const属性
const double pi=3.14159;
auto rad=pi;
//rad是一个 double类型数
const auto rad=pi;
//rad是 const double类型
```

2.4 常量修饰符和类型推导 — 类型推导

当 auto 和 decltype 遇到 const 会怎样呢?

auto

```
auto将忽略掉 const属性
const double pi=3.14159;
auto rad=pi;
//rad是一个 double类型数
const auto rad=pi;
//rad是 const double类型
```

decltype

```
decltype将不会忽略 const属性
const double pi=3.14159;
decltype (pi) rad=1.0;
//rad为 const double类型
```

2.5 表达式 — 基本知识

表达式

- 表达式是由运算符和操作对象组成的式子。如：`1 + 2 * 3`
  - 运算符是用来运算或处理对象的符号
  - 操作对象指参与运算的对象
- 表达式都有一个值，要么是左值，要么是右值
- 含有多个运算符的表达式称为复合表达式

2.5 表达式 — 基本知识

表达式

- 表达式是由运算符和操作对象组成的式子。如：`1 + 2 * 3`
  - 运算符是用来运算或处理对象的符号
  - 操作对象指参与运算的对象
- 表达式都有一个值，要么是左值，要么是右值
- 含有多个运算符的表达式称为复合表达式

运算符分类

- 根据操作数个数，分为：单目运算符、双目运算符和三目运算符
- 根据功能运算，分为：算术运算符、赋值运算符、条件运算符等

2.5 表达式 — 基本知识

左值和右值

- 左值所在的内存空间的地址是可以获取的（使用取址符 `&`），但右值的地址是无法得到的
- 左值对象由程序员创建并命名，具有持久性。右值对象中除了字面值常量以外，都是临时对象
- 一般来说，右值只能在 $=$ 符号右边，左值没有限制

2.5 表达式 — 基本知识

左值和右值

- 左值所在的内存空间的地址是可以获取的（使用取址符 &），但右值的地址是无法得到的
- 左值对象由程序员创建并命名，具有持久性。右值对象中除了字面值常量以外，都是临时对象
- 一般来说，右值只能在 = 符号右边，左值没有限制

例如

```
int i = 0;           //正确: 用右值常量 0 初始化左值对象i
10 = i;              //错误:赋值运算符左侧必须为左值
int j = i;           //正确:  读取左值对象 i的值初始化左值对象j
const int N = 100; //正确:  N 为右值对象
N = 40;              //错误: 不能改变右值对象 N 的值
```

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

例子

(1) 2*3+6/2

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

例子

(1) `2*3+6/2`

运算顺序为：`(2*3)+(6/2)`，结果为 `9`

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

例子

(1) `2*3+6/2`

运算顺序为：`(2*3)+(6/2)`，结果为 `9`

(2) `1+2+3-4`

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

例子

(1) `2*3+6/2`

运算顺序为：`(2*3)+(6/2)`，结果为 `9`

(2) `1+2+3-4`

运算次序为 `(((1+2)+3)-4)`，结果为 `2`

2.5 表达式 — 基本知识

优先级和结合性

- 优先级和结合性共同决定了运算中的优先关系
- 优先级：指不同运算符在运算中的优先关系。例如乘 (*) 和除 (/) 的优先级比加 (+) 和减 (-) 的高
- 结合性：决定优先级相等的运算符结合在一起时的运算次序，同一优先级的运算符有相等的结合性（左/右结合）。例如 + 和 - 的结合性是从左到右（左结合）

例子

(1) `2*3+6/2`

运算顺序为：`(2*3)+(6/2)`，结果为 `9`

(2) `1+2+3-4`

运算次序为 `(((1+2)+3)-4)`，结果为 `2`

建议

对于复杂的表达式求值，可以通过添加括号的方法来求解

2.5 表达式 — 算数运算符

算术运算符

运算符	名称	属性	优先级
+	正	单目，右结合	3
-	负	单目，右结合	3
*	乘	双目，左结合	5
/	除	双目，左结合	5
%	求余	双目，左结合	5
+	加	双目，左结合	6
-	减	双目，左结合	6

2.5 表达式 — 算数运算符

算术运算符

运算符	名称	属性	优先级
+	正	单目，右结合	3
-	负	单目，右结合	3
*	乘	双目，左结合	5
/	除	双目，左结合	5
%	求余	双目，左结合	5
+	加	双目，左结合	6
-	减	双目，左结合	6

优先级：加、减 < 乘、除、求余 < 单目
结果：右值

2.5 表达式 — 算数运算符

算术运算符

运算符	名称	属性	优先级
+	正	单目，右结合	3
-	负	单目，右结合	3
*	乘	双目，左结合	5
/	除	双目，左结合	5
%	求余	双目，左结合	5
+	加	双目，左结合	6
-	减	双目，左结合	6

优先级：加、减 < 乘、除、求余 < 单目
结果：右值

说明

- 整型数的算术运算的结果还是整数，例如：21/6 的结果是 3，余数被舍弃
- 求余运算中的两个运算数必须是整型类型，如：5%3
- 如果运算对象的数据类型不相同，编译器会自动进行类型转换，原则是小数据类型向大数据类型转换
- 注意数据溢出问题

2.5 表达式 — 算数运算符

思考如下表达式计算过程?

例子

```
'a' + 10 + u + d - i / f
```

其中 i 是 int 类型, u 是 unsigned int 类型, f 是 float 类型, d 是 double 类型

2.5 表达式 — 算数运算符

思考如下表达式计算过程?

例子

```
'a' + 10 + u + d - i / f
```

其中 i 是 int 类型, u 是 unsigned int 类型, f 是 float 类型, d 是 double 类型

解析

数据类型转换规则

int—>unsigned—>long—>float—>double

计算顺序

```
((((('a' + 10) + u) + d) - (i / f))
```

2.5 表达式 — 算数运算符

思考如下表达式计算过程?

例子

```
'a' + 10 + u + d - i / f
```

其中 i 是 int 类型, u 是 unsigned int 类型, f 是 float 类型, d 是 double 类型

解析

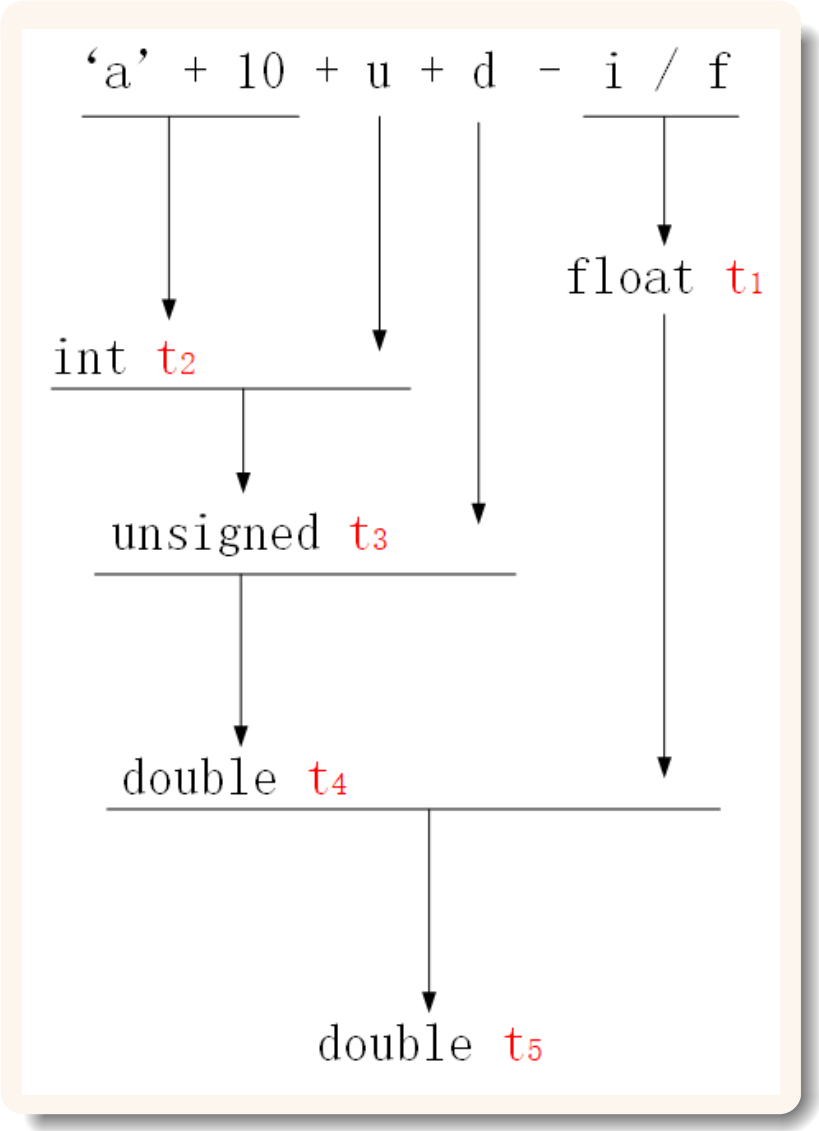
数据类型转换规则

int—>unsigned—>long—>float—>double

计算顺序

```
((((('a' + 10) + u) + d) - (i / f))
```

计算过程:



2.5 表达式 — 赋值运算符

赋值运算符：=

格式：对象 = 表达式

例如

```
int i = 2; //对象的初始化
i = 5;     //对象的赋值
i = i + 6; //对象的赋值
```

2.5 表达式 — 赋值运算符

赋值运算符：=

格式：对象 = 表达式

例如

```
int i = 2; //对象的初始化
i = 5;     //对象的赋值
i = i + 6; //对象的赋值
```

- 优先级：赋值运算符< 算术运算符
- 结合性：右结合
- 结果：左值（C 语言是右值）

2.5 表达式 — 赋值运算符

赋值运算符：=

格式：对象 = 表达式

例如

```
int i = 2; //对象的初始化
i = 5;     //对象的赋值
i = i + 6; //对象的赋值
```

- 优先级：赋值运算符< 算术运算符
- 结合性：右结合
- 结果：左值（C 语言是右值）

说明

- 功能是把赋值符号右侧表达式的值写入到赋值符号左侧的操作对象里面
- 左侧操作对象必须是支持写操作的左值，例如

```
int i = 0, j = 2;
i + j = 10;//错误：算术表达式为右值
```

2.5 表达式 — 赋值运算符

注意

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

```
int i = 0;
double d = 3.14159;
i = d; //表达式的结果的类型是 int, 值是3, 但d的值不变
```

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

```
int i = 0;
double d = 3.14159;
i = d; //表达式的结果的类型是 int，值是3，但d的值不变
```

- 赋值运算满足右结合性，例如

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

```
int i = 0;
double d = 3.14159;
i = d; //表达式的结果的类型是 int，值是3，但d的值不变
```

- 赋值运算满足右结合性，例如

```
i = j = 5; //i 和 j 的值都是5
//根据右结合性，先把 5 赋给 j ，然后把 j = 5 的值赋给 i。
```

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

```
int i = 0;
double d = 3.14159;
i = d; //表达式的结果的类型是 int，值是3，但d的值不变
```

- 赋值运算满足右结合性，例如

```
i = j = 5; //i 和 j 的值都是5
//根据右结合性，先把 5 赋给 j ，然后把 j = 5 的值赋给 i。
```

- 赋值运算符的优先级是较低，有时需要加上括号才能正确执行，例如：

2.5 表达式 — 赋值运算符

注意

- 对象的初始化不是赋值操作，赋值是对象已经定义的情况下，赋一个新值
- 两侧的操作对象的类型不相同，会进行类型转换，但右侧本身的不会发生变化，如

```
int i = 0;
double d = 3.14159;
i = d; //表达式的结果的类型是 int，值是3，但d的值不变
```

- 赋值运算满足右结合性，例如

```
i = j = 5; //i 和 j 的值都是5
//根据右结合性，先把 5 赋给 j ，然后把 j = 5 的值赋给 i。
```

- 赋值运算符的优先级是较低，有时需要加上括号才能正确执行，例如：

```
i = 2 + j = 4; //错误：2 + j 为右值，不能作为第二个赋值符号的左侧对象
i = 2 + (j = 4); //正确：把 j = 4 的值加上2赋给 i ， i 和 j 的值分别为6和4
```

2.5 表达式 — 赋值运算符

代码示例

```
int counter;
counter = counter + 5;
```

2.5 表达式 — 赋值运算符

代码示例

```
int counter;
counter = counter + 5;
```

代码分析

第二行代码需要进行两次计算，首先计算 (couter+5) 的值，然后再赋值给 counter。为简化计算，C++ 提供了复合运算符 (+=　-=　/=　*=　%=)

2.5 表达式 — 赋值运算符

代码示例

```
int counter;
counter = counter + 5;
```

例如

```
counter += 1;
//等效于 counter = counter + 1;
//且只需计算一次，提高了计算效率
i *= j + 3;
//等效于 i = i * (j + 3);
```

代码分析

第二行代码需要进行两次计算，首先计算 (couter+5) 的值，然后再赋值给 counter。为简化计算，C++ 提供了复合运算符 (+=　-=　/=　*=　%=)

2.5 表达式 — 赋值运算符

代码示例

```
int counter;
counter = counter + 5;
```

例如

```
counter += 1;
//等效于 counter = counter + 1;
//且只需计算一次，提高了计算效率
i *= j + 3;
//等效于 i = i * (j + 3);
```

代码分析

第二行代码需要进行两次计算，首先计算 (couter+5) 的值，然后再赋值给 counter。为简化计算，C++ 提供了复合运算符 (+=　-=　/=　*=　%=)

小结

使用复合赋值运算符能够提高运算效率，减少了产生临时对象这一环节

2.5 表达式 — 自增自减运算符

观察如下代码

代码示例

```
int i = 5, j = 6;
i += 1;
j -= 1;
```

2.5 表达式 — 自增自减运算符

观察如下代码

代码示例

```
int i = 5, j = 6;
i += 1;
j -= 1;
```

为简化自增 1 和自减 1 操作，C++ 提供了自增（++）和自减（--）运算符

2.5 表达式 — 自增自减运算符

观察如下代码

代码示例

```
int i = 5, j = 6;
i += 1;
j -= 1;
```

为简化自增 1 和自减 1 操作，C++ 提供了自增（++）和自减（--）运算符

例如

```
int i = 0, j;
j = i++;
//后置，i的值自增变为1，表达式 i++ 的值为 i 自增之
  前的值，即 j 的值为0
j = ++i;
//前置，i的值自增变为2，表达式 ++i 的值为 i 自增之
  后的值，即 j 的值为2
```

2.5 表达式 — 自增自减运算符

观察如下代码

代码示例

```
int i = 5, j = 6;
i += 1;
j -= 1;
```

例如

```
int i = 0, j;
j = i++;
//后置，i的值自增变为1，表达式 i++ 的值为 i 自增之
  前的值，即 j 的值为0
j = ++i;
//前置，i的值自增变为2，表达式 ++i 的值为 i 自增之
  后的值，即 j 的值为2
```

为简化自增 1 和自减 1 操作，C++ 提供了自增（++）和自减（--）运算符

注意

- 自增、自减运算符的操作对象必须为左值
- 前置版本返回左值对象本身，后置版本将原始值的副本作为右值返回
- 建议能用前置版本不用后置版本

2.5 表达式 — 逻辑和关系运算符

逻辑和关系运算符

运算符	功能	结合性	用法
!	逻辑非	右	!5
<	小于	左	5<4
<=	小于等于	左	5<=4
>	大于	左	5>4
>=	大于等于	左	5>=4
==	等于	左	5==4
!=	不等于	左	5!=4
&&	逻辑与	左	5&&4
||	逻辑或	左	5||4

2.5 表达式 — 逻辑和关系运算符

逻辑和关系运算符

运算符	功能	结合性	用法
!	逻辑非	右	!5
<	小于	左	5<4
<=	小于等于	左	5<=4
>	大于	左	5>4
>=	大于等于	左	5>=4
==	等于	左	5==4
!=	不等于	左	5!=4
&&	逻辑与	左	5&&4
||	逻辑或	左	5||4

优先级：
赋值运算符 < 逻辑与 && 、逻辑或 || < 关系运算符 < 算术运算符 < 逻辑非!

2.5 表达式 — 逻辑和关系运算符

逻辑和关系运算符

运算符	功能	结合性	用法
!	逻辑非	右	!5
<	小于	左	5<4
<=	小于等于	左	5<=4
>	大于	左	5>4
>=	大于等于	左	5>=4
==	等于	左	5==4
!=	不等于	左	5!=4
&&	逻辑与	左	5&&4
||	逻辑或	左	5||4

优先级：
赋值运算符 < 逻辑与 && 、逻辑或 || < 关系运算符 < 算术运算符 < 逻辑非!

关系运算符内部优先级：
== 和 != 低于 <=、> 和 >=

结果：右值

2.5 表达式 — 逻辑和关系运算符

可以用 `i <= j <= k` 直接构造表达式$i \leq j \leq k$的逻辑关系吗?

2.5 表达式 — 逻辑和关系运算符

可以用 `i <= j <= k` 直接构造表达式$i \leq j \leq k$的逻辑关系吗?

答案

不能，因为只要 `k` 大于等于 `1`，表达式 `i <= j <= k` 的值永远为真，应该使用逻辑与 `&&` 构造表达式，即：`i <= j && j <= k`

2.5 表达式 — 逻辑和关系运算符

可以用 `i <= j <= k` 直接构造表达式$i \leq j \leq k$的逻辑关系吗?

答案

不能，因为只要 `k` 大于等于 `1`，表达式 `i <= j <= k` 的值永远为真，应该使用逻辑与 `&&` 构造表达式，即：`i <= j && j <= k`

2.5 表达式 — 逻辑和关系运算符

给出以下代码运行后 `j` 和 `b` 的值

示例代码

```
int i = 1, j = 2;
bool b = !i && ++j;
```

2.5 表达式 — 逻辑和关系运算符

给出以下代码运行后 `j` 和 `b` 的值

示例代码

```
int i = 1, j = 2;
bool b = !i && ++j;
```

答案

j=2, b=0

2.5 表达式 — 逻辑和关系运算符

给出以下代码运行后 j 和 b 的值

示例代码

```
int i = 1, j = 2;
bool b = !i && ++j;
```

答案

j=2, b=0

&& 和 || 运算符的运算规则

逻辑与 &&和逻辑或 ||运算符都是先计算左侧对象的值，然后根据左侧对象的值判断是否计算右侧运算对象的值，即为短路求值

- &&，仅当左侧运算对象的值为真时，才计算右侧运算对象的值
- ||，仅当左侧运算对象的值为假时，才计算右侧运算对象的值

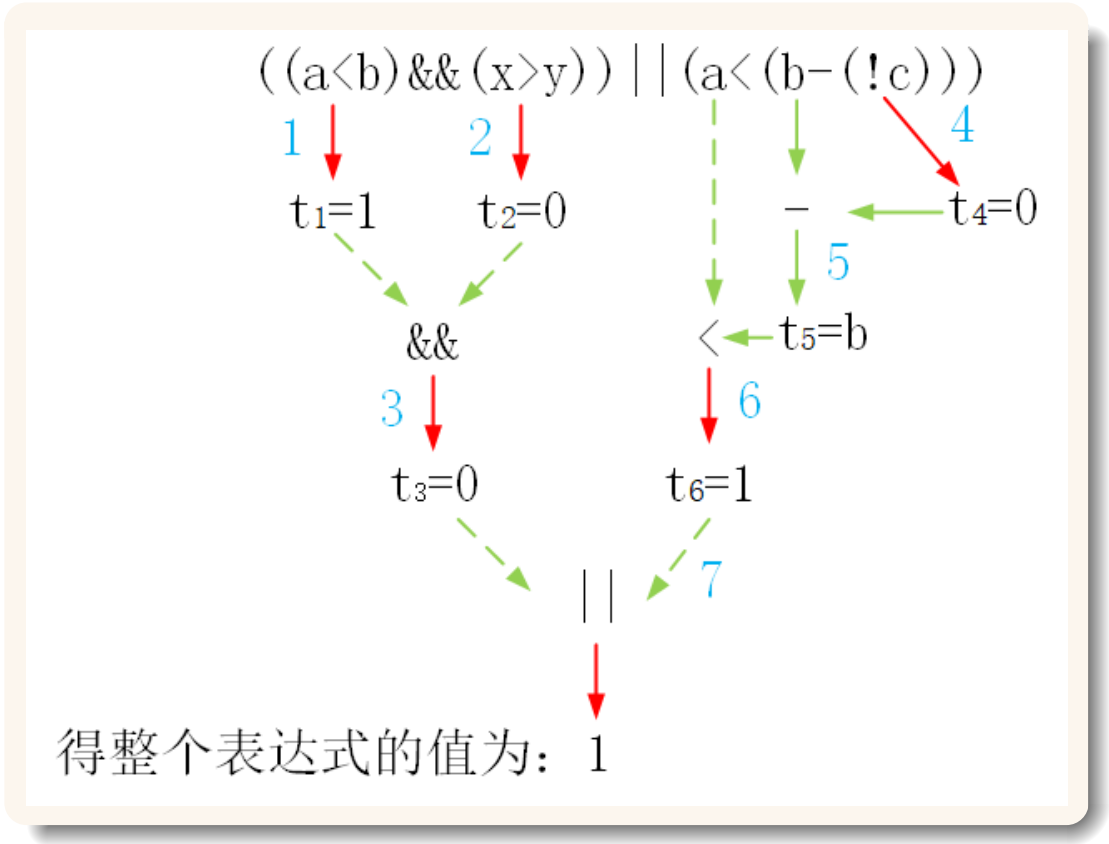
2.5 表达式 — 逻辑和关系运算符

已知 `int a=10, b=20, c=30; float x=1.8, y=2.4;`
求解 `a<b&&x>y||a<b-!c`

2.5 表达式 — 逻辑和关系运算符

已知 `int a=10, b=20, c=30; float x=1.8, y=2.4;`
求解 `a<b&&x>y||a<b-!c`

答案：



2.5 表达式

练习

1. 已知 `int a=1,b=5,c=2,d=6;` 求解下列表达式及相应对象的的值

   (1)`a+b>c+d`　　(2)`a=(b=4)+(c=6)`

   (3)`a+=c=b*a`　　(4)`d=(a++)-(++b)+c--`

   (5)`--a||b<d&&c--`

2. 逻辑表达式的构造：`a` 和 `b` 之一为 `0`，但不同时为 `0`

2.5 表达式

练习

1. 已知 `int a=1,b=5,c=2,d=6;` 求解下列表达式及相应对象的的值

(1)`a+b>c+d`　　(2)`a=(b=4)+(c=6)`

(3)`a+=c=b*a`　　(4)`d=(a++)-(++b)+c--`

(5)`--a||b<d&&c--`

2. 逻辑表达式的构造：`a` 和 `b` 之一为 `0`，但不同时为 `0`

1.(1) 表达式的值为 0，a,b,c,d 不变

(2) 表达式的值为 10，a=10,b=4,c=6

(3) 表达式的值为 6，a=6,b=5,c=5

(4) 表达式的值为-3，a=2,b=6,c=1,d=-3

(5) 表达式的值为 1，a=0,b=5,c=1,d=6

2. 答案一：`a==0&&b!=0||a!=0&&==0`

答案二：`a*b==0&&a+b!=0`

2.5 表达式 — 逗号运算符

逗号表达式：由逗号运算符连接起来的表达式。从左向右依次计算每个运算对象，结果为最右边的运算对象

格式如下

```
exp1, exp2, ...
```

2.5 表达式 — 逗号运算符

逗号表达式：由逗号运算符连接起来的表达式。从左向右依次计算每个运算对象，结果为最右边的运算对象

格式如下

```
exp1, exp2, ...
```

例如

```
int i, j;
i = (j=3, j+=6, 5+6);
// i 的值为11, j 的值为9
```

2.5 表达式 — 逗号运算符

逗号表达式：由逗号运算符连接起来的表达式。从左向右依次计算每个运算对象，结果为最右边的运算对象

格式如下

```
exp1, exp2, ...
```

优先级：

逗号运算符< 赋值运算符 < 关系运算符 < 算术运算符

例如

```
int i, j;
i = (j=3, j+=6, 5+6);
// i 的值为11, j 的值为9
```

2.5 表达式 — 逗号运算符

逗号表达式：由逗号运算符连接起来的表达式。从左向右依次计算每个运算对象，结果为最右边的运算对象

格式如下

```
exp1, exp2, ...
```

优先级：

逗号运算符< 赋值运算符 < 关系运算符 < 算术运算符

例如

```
int i, j;
i = (j=3, j+=6, 5+6);
// i 的值为11, j 的值为9
```

说明

- 逗号表达式的值是最右边表达式的值
- 在所有的运算符中，逗号运算符的优先级最低
- 左侧代码中，i 的值为 11，j 的值为 9

2.5 表达式 — 条件运算符

条件运算符 (?:) 唯一的三目运算符

格式如下

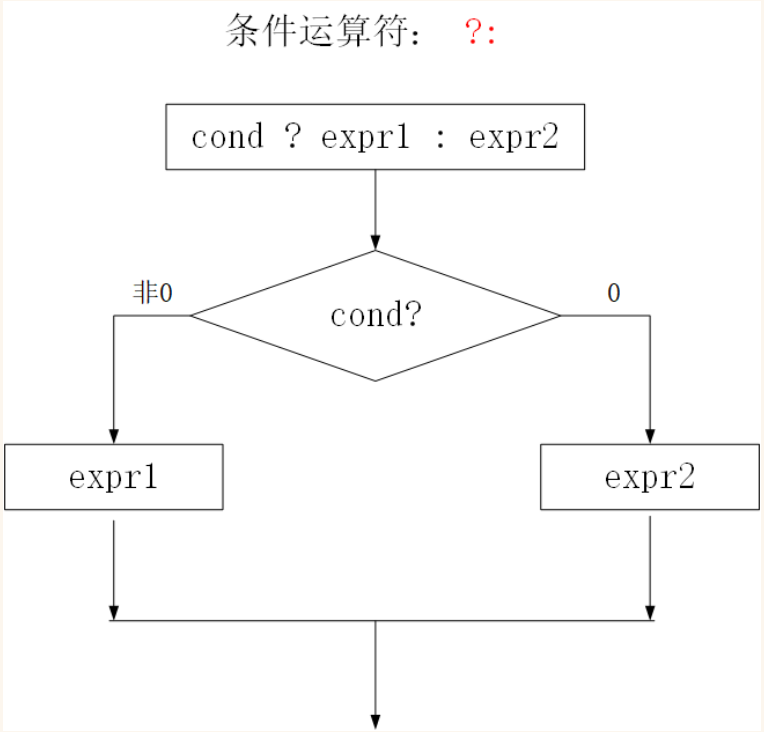
```
cond ? expr1 : expr2
```

2.5 表达式 — 条件运算符

条件运算符 (?:) 唯一的三目运算符

格式如下

```
cond ? expr1 : expr2
```

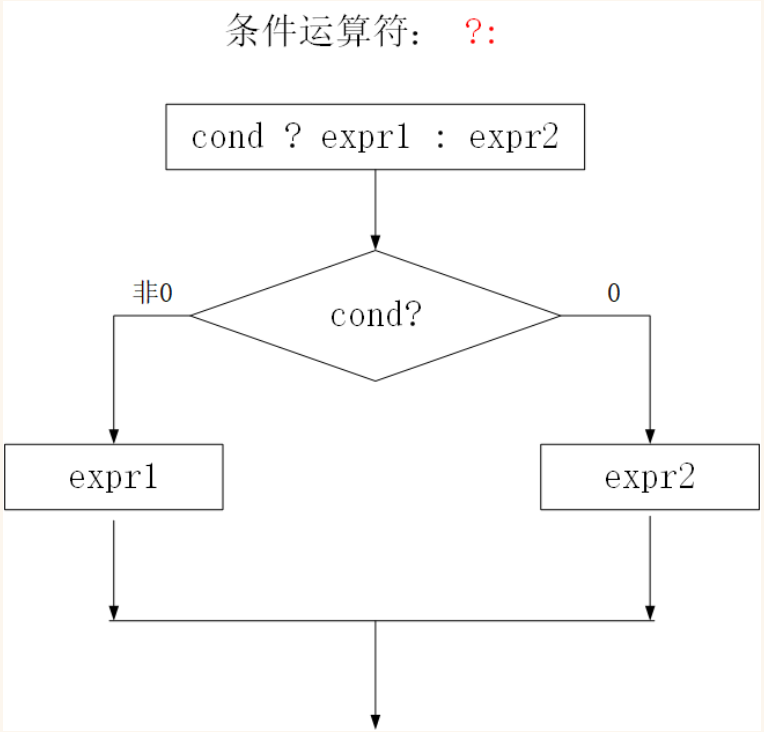


2.5 表达式 — 条件运算符

条件运算符 (?:) 唯一的三目运算符

格式如下

```
cond ? expr1 : expr2
```



条件运算符允许嵌套使用，如

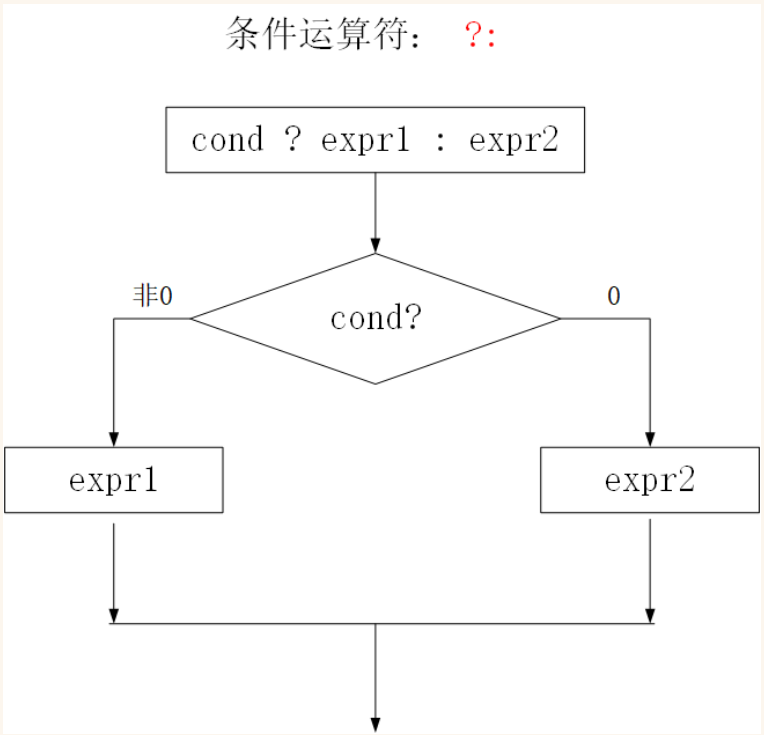
```
int a=4, b=5, c=6, max;
max=a>b?(a>c?a:c):(b>c?b:c);
```

2.5 表达式 — 条件运算符

条件运算符 (?:) 唯一的三目运算符

格式如下

```
cond ? expr1 : expr2
```



条件运算符允许嵌套使用，如

```
int a=4, b=5, c=6, max;
max=a>b?(a>c?a:c):(b>c?b:c);
```

建议

由于会降低程序的可读性，条件运算符不宜嵌套使用

2.5 表达式 — sizeof 运算符

`sizeof` 运算符返回一个表达式或一个类型所占内存的字节数

一般格式

格式:

```
sizeof (type) 或
sizeof (expr)
```

例如:

```
cout << sizeof (int);//输出4
int i=0;
cout << sizeof (++i);//输出4, i 的值为0
```

2.5 表达式 — sizeof 运算符

sizeof 运算符返回一个表达式或一个类型所占内存的字节数

一般格式

格式:
```
sizeof (type) 或
sizeof (expr)
```
例如:
```
cout << sizeof (int);//输出4
int i=0;
cout << sizeof (++i);//输出4, i 的值为0
```

注意

sizeof(expr) 形式只是返回表达式结果的数据类型的字节数，并不会实际运算表达式，因此左边执行完输出语句后，i 的值仍然为 0

2.5 表达式 — 位运算符

- 位运算符： ~ 、<< 、>> 、& 、| 、 ^ (优先级由高到低)
- 运算的对象是整型对象，处理二进制数

2.5 表达式 — 位运算符

- 位运算符： ~ 、<< 、>> 、& 、| 、 ^ (优先级由高到低)
- 运算的对象是整型对象，处理二进制数

```
short a = 3, b = 5;
```

b	~	00000000 00000101	<<1	00000000 00000101	>>1	00000000 00000101
		11111111 11111010		00000000 00001010		00000000 00000010
a		00000000 00000011		00000000 00000011		00000000 00000011
b	&	00000000 00000101	|	00000000 00000101	^	00000000 00000101
		00000000 00000001		00000000 00000111		00000000 00000110

2.5 表达式 — 求值次序

求值次序

示例代码

```
int i = 0, j;
j = i * 2 + i++;
```

问题

是先计算 `i++` 还是 `i*2`，如果先计算表达式 `i*2` 再计算表达式 `i++`，那么结果为 `0`；反过来，结果则为 `2`?

2.5 表达式 — 求值次序

求值次序

示例代码

```
int i = 0, j;
j = i * 2 + i++;
```

测试结果

- VS 编译器，j 的值为 0
- GCC 编译器，j 的值为 2

问题

是先计算 i++ 还是 i*2，如果先计算表达式 i*2 再计算表达式 i++，那么结果为 0；反过来，结果则为 2?

2.5 表达式 — 求值次序

求值次序

示例代码

```
int i = 0, j;
j = i * 2 + i++;
```

测试结果

- VS 编译器，j 的值为 0
- GCC 编译器，j 的值为 2

问题

是先计算 i++ 还是 i*2，如果先计算表达式 i*2 再计算表达式 i++，那么结果为 0；反过来，结果则为 2?

建议

在复合表达式中，不要出现对同一个对象既读又写的情况，容易出错

2.6 类型转换 — 隐式转换

类型转换 { 隐式转换
显示转换（强制类型转化）

2.6 类型转换 — 隐式转换

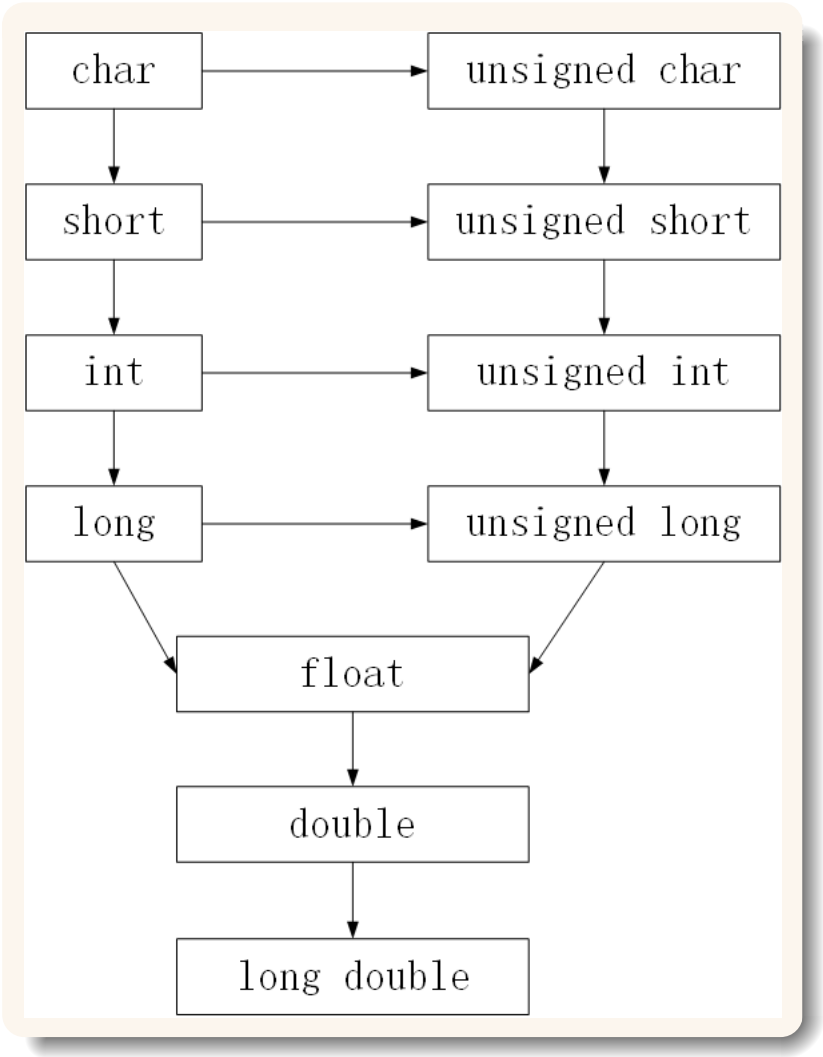
类型转换 { 隐式转换
显示转换（强制类型转化）

隐式转换

- 比 int 类型小的整型类型提升为较大的整型类型
- 将表达式的值转换为布尔值
- 初始化过程中，初始值转换成定义对象的类型
- 算术表达式中，运算结果转换为运算对象中最宽（大）的数据类型

2.6 类型转换 — 隐式转换

混合运算的类型转换规则



2.6 类型转换 — 显示转换

显示转换方式:

- `static_cast`、`dynamic_cast`、`const_cast`、`reinterpret_cast`

2.6 类型转换 — 显示转换

显示转换方式:

- `static_cast`、`dynamic_cast`、`const_cast`、`reinterpret_cast`

几种转换方式的应用

- `static_cast` 在算术表达式中的应用

2.6 类型转换 — 显示转换

显示转换方式:

- static_cast、dynamic_cast、const_cast、reinterpret_cast

几种转换方式的应用

- static_cast 在算术表达式中的应用
  - 执行浮点数操作，如:

```
int i = 5, j = 3;
double k = i / static_cast<double>(j); //强制将 j 转化为 double 类型
```

2.6 类型转换 — 显示转换

显示转换方式:

- static_cast、dynamic_cast、const_cast、reinterpret_cast

几种转换方式的应用

- static_cast 在算术表达式中的应用
  - 执行浮点数操作，如:

```
int i = 5, j = 3;
double k = i / static_cast<double>(j); //强制将 j 转化为 double 类型
```

  - 告诉编译器我们有意将宽类型转换成窄类型，请关闭警告信息，例如:

```
double i = 5., j = 3.;
int k = static_cast<int>(i / j);//强制将 i / j 的结果转化为 int
```

2.6 类型转换 — 显示转换

显示转换方式:

- static_cast、dynamic_cast、const_cast、reinterpret_cast

几种转换方式的应用

- static_cast 在算术表达式中的应用
  - 执行浮点数操作，如:

```
int i = 5, j = 3;
double k = i / static_cast<double>(j); //强制将 j 转化为 double 类型
```

  - 告诉编译器我们有意将宽类型转换成窄类型，请关闭警告信息，例如:

```
double i = 5., j = 3.;
int k = static_cast<int>(i / j);//强制将 i / j 的结果转化为 int
```

- const_cast 常用来去掉对象的 const 属性，即把 const 对象转换为非 const 对象

2.6 类型转换 — 显示转换

其他的类型转换方式

例如

格式:

```
type (expr) //函数方式, 或者
(type) expr //C语言方式
```

例如:

```
double k = i / (double)j;
//强制将 j 转化为 double 类型
double k = i / double (j);
```

2.6 类型转换 — 显示转换

其他的类型转换方式

例如

格式:

```
type (expr) //函数方式，或者
(type) expr //C语言方式
```

例如:

```
double k = i / (double)j;
//强制将 j 转化为 double 类型
double k = i / double (j);
```

提示

类型转换不会改变对象本身的值，如:

```
double i = 5.;
int j = i;
//将 i的结果转化为 int,但i的值不变
int k = static_cast<int>(i);
//强制将 i的结果转化为 int,但i的值不变
```

本章结束