1 从C到C++

1.1 背景介绍

1.1.1 C++是一种编程语言

- 编程语言排行

Position Nov 2012	Position Nov 2011	Delta in Position	Programming Language	Ratings Nov 2012	Delta Nov 2011	Status
1	2	↑	C	19.224%	+1.90%	A
2	1	↓	Java	17.455%	-0.42%	A
3	6	↑↑↑	Objective-C	10.383%	+4.40%	A
4	3	↓	C++	9.698%	+1.61%	A
5	5	=	PHP	5.732%	-0.36%	A
6	4	↓↓	C#	5.591%	-1.73%	A
7	7	=	(Visual) Basic	5.032%	-0.01%	A
8	8	=	Python	4.062%	+0.45%	A
9	10	↑	Perl	2.182%	+0.10%	A
10	11	↑	Ruby	1.739%	+0.24%	A
11	9	↓↓	JavaScript	1.278%	-1.29%	A
12	16	↑↑↑↑	Delphi/Object Pascal	0.995%	+0.12%	A
13	13	=	Lisp	0.951%	-0.23%	A
14	14	=	Pascal	0.881%	-0.11%	A
15	23	↑↑↑↑↑↑↑↑	Visual Basic .NET	0.769%	+0.24%	A-
16	19	↑↑↑	Ada	0.662%	+0.04%	B
17	12	↓↓↓↓↓	PL/SQL	0.632%	-0.81%	B
18	18	=	Lua	0.631%	0.00%	A-
19	15	↓↓↓↓	MATLAB	0.620%	-0.34%	B
20	24	↑↑↑↑	Assembly	0.585%	+0.06%	B

- 编程语言演变

A QUICK LOOK AT PROGRAMMING LANGUAGES

YEAR: 1957
LANGUAGE: FORTRAN

Tim Jenner / Shutterstock.com

FORmula**TRAN**slation, is the oldest language still in use. Created by John Backus, the language was developed to perform high-level scientific, mathematical, statistical computations.

The language is still used in aerospace, automotive industries, government, and research institutions.

» Used by NATIONAL WEATHER SERVICE

A LOOK AT THE CODE:

```
*
C Hello World in Fortran 77
C (lines must be 6 characters
indented)
*
PROGRAM REELECT
WRITE(UNIT=*, FMT=*)
'I like Ike'
END
```

Created in 1874 to increase typing speed, the QWERTY keyboard was responsible for the majority of computer languages ever created.

The arrangement was set based on an analysis of keys likely to cause jams and designing the QWERTY board to separate likely offenders, like T and H.

YEAR: 1959
LANGUAGE: COBOL

COmmon **B**usiness **O**riented **L**anguage is behind the majority of business transaction systems running credit card processing, ATMs, telephone and cell calls, hospital systems, government, automotive systems, and traffic signal systems. The COBOL development team, lead by Dr. Grace Murray Hopper, set out to create a uniform, user-friendly language for business transactions.

UNITED STATES POSTAL SERVICE. » Used by UNITED STATES POSTAL SERVICE

A LOOK AT THE CODE:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
StandardAlert.
AUTHOR. Fabritius.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
PROCEDURE DIVISION.
DISPLAY "DANGER! DESTROY!".
STOP RUN.
```

In 1937, Binary Code was Claude Shannon's thesis on translating text into mathematical code was the foundation for the first fully operational electromechanical computer, Zuse's 1941 Z3.

Computers still speak binary code, but most modern programmers never touch the 0s and 1s.

YEAR: 1964
LANGUAGE: BASIC

Developed by students at Dartmouth College, Beginners All Purpose Symbolic Instruction Code was designed to be a simplified language for those without a strong technical or mathematical background. A modified version, written by Bill Gates and Paul Allen became Microsoft's first product. It was sold to M.I.T.S. for the Altair.

» INTEGER BASIC RAN THE ORIGINAL APPLE II IN 1977

A LOOK AT THE CODE:

```
100 BEGIN
101 GOTO 102
102 PRINT "HOW ABOUT
A NICE GAME OF CHESS?"
103 END
```

Basic has over 2 million lines of code in use, in 1975 there were only 4,000.

YEAR: 1969
LANGUAGE: C

C was developed between 1969 and 1973 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system. It was named "C" because its features were derived from an earlier language called "B."

C had become powerful enough that most of the Unix kernel was rewritten in C - one of the first operating system kernels implemented in a language other than assembly.

» LINUX TODAY IS BASED ON C

A LOOK AT THE CODE:

```
#include <stdio.h>

main()
{
  puts ("your first C
program");
}
```

YEAR: 1970
LANGUAGE: PASCAL

The language was named for Blaise Pascal, credited for inventing the first adding machine in 1641. Niklaus Wirth created Pascal as a teaching tool and it grew to into widespread commercial use.

» Used by SKYPE (OBJECT PASCAL)

A LOOK AT THE CODE:

```
PROGRAM SageAdvice (OUTPUT);
BEGIN
WRITELN('If you spent all those
hours');
WRITELN('Learning C instead of
Pascal');
WRITELN('you might have a job
now.');
END.
```

The first version of Word had 27,000 lines of code. Today, the current version of Office has over 30 million.

YEAR: 1983
LANGUAGE: C++

From Bell Labs, Bjarne Stroustrup modified the C language to C++ and created what many consider the most popular programming language ever. It's been listed in the top ten programming languages since 1986 and achieved Hall of Fame status in 2003.

» Used by MS OFFICE; ADOBE PDF READER; FIREFOX

A LOOK AT THE CODE:

```
#include<iostream>
using namespace std;
int main()
{
cout<< "C++ is the grade
you get when you're very,
very slightly above
average." <<endl;
return 0;
}
```

YEAR: 1987
LANGUAGE: PERL

Larry Wall, a UNIX programmer, created Perl after attempting to extract data for a report and finding UNIX couldn't perform the operations he needed. Practical Extraction Report Language was described by its inventor as a language for "getting your job done."

» Used by CRAIGSLIST

A LOOK AT THE CODE:

```
#!/usr/bin/perl
# Hello World in Perl
print "I don't always
write convoluted
scripts, but when I do,
I write them in Perl.
\n";
```

YEAR: 1991

LANGUAGE: PYTHON

Monty Python served as the inspiration for the name of this language. Guido Van Rossum developed Python to fix problems in the ABC language and continues to serve as its lead designer.

YouTube » Used by GOOGLE SEARCH, YOUTUBE, NASA

A LOOK AT THE CODE:

```
# Hello World in Python
print"The airspeed velocity of
an unladen European Swallow is
approximately 11 meters per
second."
```

The Mac OS/X uses about 90 million lines of code.

YEAR: 1993
LANGUAGE: RUBY

Yukihiro "matz" Matsumoto named Ruby for July's birthstone. HE developed the language by blending parts of his favorite languages, Perl, Smalltalk, Eiffel, Ada, and Lisp.

» Used by BASECAMP

A LOOK AT THE CODE:

```
"I'd rather be
writing this in
Java.\n".display
```

YEAR: 1995
LANGUAGE: PHP

Rasmus Lerdorf developed PHP to replace a set Perl scripts used to maintain his personal home page. Today, PHP has grown in to an integral part of web architecture running on over 20 million websites.

» Used by FACEBOOK

A LOOK AT THE CODE:

```
<?php
echo"Fun Fact:
PHP used to
stand for
'Personal Home
```

A mere 15 million lines of code ran Windows 95. Windows 7 uses more than 50 million

```
?>
```

lines of code.

YEAR: 1995
LANGUAGE: JAVA

A team of Sun Microsystems developers lead by James Gosling created Java to run set top boxes for interactive television. Java now runs on over 1.1 billion PCs worldwide and many websites can't function with out it.

» Used by 2004 MARS ROVERS

A LOOK AT THE CODE:

```
public class HelloWorld {
public static void
main(String[] args) {
System.out.println("I
will now interrupt you
with update notifications
every other day for the
rest of your life.");
}
}
```

YEAR: 1995
LANGUAGE: JAVASCRIPT

Java and Javacript are unrelated and have very different semantics.

JavaScript was originally developed by Brendan Eich of Netscape under the name Mocha. JavaScript uses syntax influenced by that of C.

Although meant to run on the client (browser) it is now finding use on the server as node.js. Also, AJAX is dependent on Javascript.

rackspace HOSTING » Used by RACKSPACE (CLIENT SIDE)

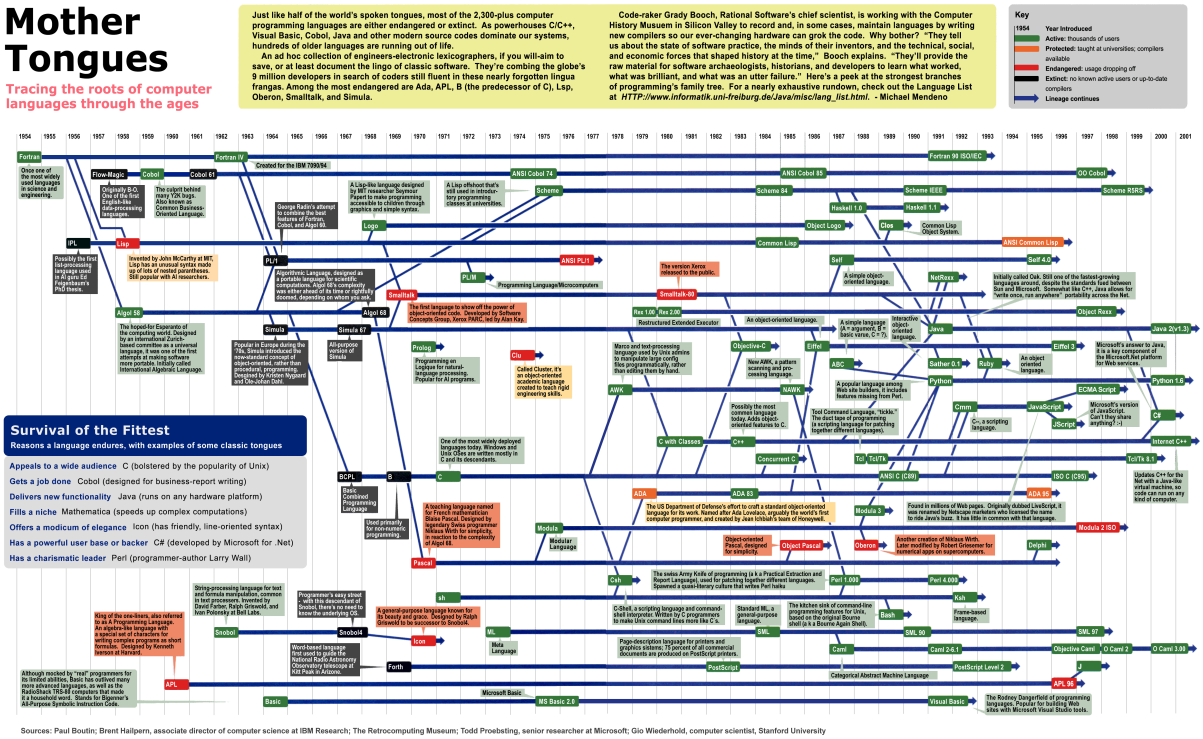
A LOOK AT THE CODE:

```
<html>
<body>

<script type="text/javascript">
document.write("<h1>This is a
heading</h1>");
document.write("<p>This is a
paragraph.</p>");
document.write("<p>This is
another paragraph.</p>");
</script>

</body>
</html>
```

YEAR: 2005
LANGUAGE: RUBY ON RAILS

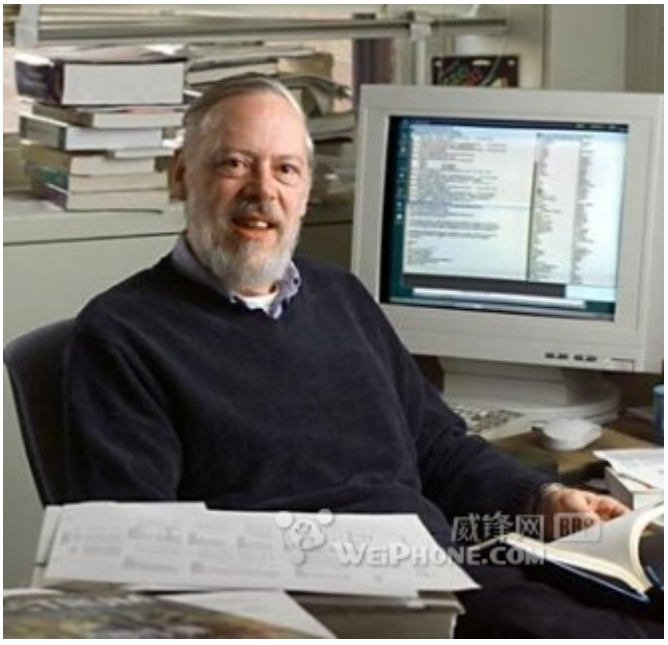

LANGUAGE: RUBY ON RAILS
(Framework for programming language, Ruby)
Ruby on Rails was extracted by David Heinemeier Hansson from his work on Basecamp, a project management tool by 37signals. Hansson first released Ruby on Rails as open source in July 2004, but did not share commit rights to the project until February 2005. It is now on version 3.0.7 and has more than 1,800 contributors.
A single game application for the iPhone uses around 2 million lines of code.
A LOOK AT THE CODE:
def section_link(name,options) if options[:action] == @current_action and options[:controller] == @current_controller link_to(name, options, :class => 'on') else link_to(name,options) end end
PROGRAMMING LANGUAGE POPULARITY
MAY 2011
RPG
SCHEME
TRANSACT/SQL
PASCAL
ADA
LISP
ASSEMBLY
DELPHI
LUA
JAVASCRIPT
RUBY
PERL
VISUAL (BASIC)
PYTHON
OBJECTIVE-C
PHP
C#
C++
C
JAVA
0%
5%
10%
15%
20%

- 编程语言图谱



1.1.2 致敬与缅怀

- Dennis M. Ritchie（丹尼斯·里奇），1941-2011，C语言之父、UNIX之父、黑客之父



- Ken Thompson（肯·汤普逊），1943-，B语言之父、UNIX的发明人之一

- Bjarne Stroustrup（本贾尼·斯特劳斯特卢普），1950-，C++之父

- C++之父的个人主页：http://www2.research.att.com/~bs/homepage.html

1.1.3 C++之父的伟大贡献

- Bjarne Stroustrup，1950年出生于丹麦，先后毕业于丹麦阿鲁斯大学和英国剑桥大学，AT&T大规模程序设计研究部门负责人，AT&T、贝尔实验室和ACM成员
- 最初导致C++诞生的原因是在Bjarne博士等人试图去分析UNIX内核的时候，这项工作开始于1979年4月，当时由于没有合适的工具能够有效地分析由于内核分布而造成的网络流量，以及怎样将内核模块化。同年10月，Bjarne博士完成了一个可以运行的预处理程序，称之为Cpre，它为C加上了类似Simula的类机制。在这个过程中，Bjarne博士开始思考是不是要开发一种新的语言，当时贝尔实验室对这个想法很感兴趣，就让Bjarne博士等人组成一个开发小组，专门进行研究
- 当时的C++还不叫C++，而是C with class，这是把它当作一种C语言的有效扩充。由于当时C语言在编程界居于老大的地位，要想发展一种新的语言，最强大的竞争对手就是C语言，所以当时有两个问题最受关注：C++要在运行时间、代码紧凑性和数据紧凑性方面能够与C语言相媲美，同时还要尽量避免在语言应用领域方面的限制。在这种情况下，一个很自然的想法就是让C++从C语言继承过来，但是Bjarne博士更具有先见之明，他为了避免受到C语言的局限，参考了很多语言，例如：从Simula继承了类的概念，从Algol68继承了运算符重载、引用以及在任何地方声明变量的能力，从BCPL获得了//注释，从Ada得到了模板、名字空间，从Ada、Clu和ML取来了异常
- 直到1998年，ANSI/ISO C++标准建立，同年，Bjarne博士推出了其经典著作《The C++ Programming Language》第三版。C++的标准化标志着Bjarne博士倾20年心血的伟大构想终于实现

1.1.4 C++大事记

- 1979：Bjarne博士在贝尔实验室完成了一个预处理程序——Cpre——为C加上了类似Simula的类机制
- 1983：第一个C with class实现投入使用，Rick Mascitti将其命名为C++
- 1985：CFront 1.0发布。Bjarne博士完成了经典巨著《The C++ Programming Language》第一版

- 1987：GNU C++发布
- 1990：Borland C++发布
- 1992：Microsoft C++发布。IBM C++发布
- 1998：ISO标准被批准，即C++98
- 2011：ISO发布ISO/IEC 14882:2011，即C++11
- 2014至今：ISO陆续发布C++14、C++17和C++20标准，C++23标准的正式版本即将发布

1.1.5 C++的里程碑标准

- ISO/IEC 14882，正式发布于1998年，并于2003年更新，通常称作C++98/03。目前所有主流编译器均支持此标准
- ISO/IEC 14882:2011，正式发布于2011年9月1日，官方名为Information technology - Programming languages - C++ Edition:3，即C++11，曾被临时命名为C++0x。目前绝大多数编译器都支持此标准
    - 这是1998年以来C++语言的第一次大修订，对C++语言进行了改进和扩充，新的特性也扩展了语言在灵活性和效率上的传统优势
    - 官网链接：http://www.iso.org/iso/pressrelease.htm?refid=Ref1472
    - Bjarne博士在自己网站上提供了该标准的草案文本，与最终的正式版本差别不大
- C++14/17/20，在C++11的基础上，进一步扩充新的语法特性和标准库。目前只有部分编译器支持这些标准的部分特性

1.1.6 C++的尴尬境遇

- 一方面在企业级系统(数据密集、业务规则复杂多变)开发中，C++已经基本被Java和C#等淘汰出局，另一方面在系统编程和嵌入式等更接近硬件的领域，又遭到C的强烈狙击
- 如果在你的应用中，有两个需求同时发生，你就必须要考虑采用C++，第一是对性能的要求高，第二是要有很强的抽象和建模能力

1.1.7 C++的适用领域

- 游戏：C++的效率是一个很重要的原因
- 科学计算：在科学计算领域，FORTRAN是使用最多的语言之一。但是近年来，C++凭借先进的数值计算库、泛型编程等优势在这一领域也应用颇多
- 网络和分布式应用：C++拥有很多成熟的用于网络通信的库，其中最具有代表性的是跨平台的、重量级的ACE库，该库可以说是C++语言最重要的成果之一，在许多重要的企业、部门甚至军方项目中都有应用
- 操作系统和设备驱动：在该领域，C语言是主要使用的编程语言。但是C++凭借其对C的兼容性和面向对象特性，也开始在该领域崭露头角
- 移动终端、嵌入式系统、教育科研、部分行业应用：C++、Java和C#这3种语言中，C++是最早出现的，保持了对C的兼容性，允许指针的存在，允许程序员手动高效地管理和使用内存（尽管这也是最容易引起问题的地方）

1.1.8 C++还是++C?

- 后缀++表达式的值，是变量自增前的值，这是否意味着广大C++程序员们，还在按照C的方式使用C++呢?

1.2 更好的C

1.2.1 语言风格更简洁

- C语言

```
/* minisdk.cpp */

#include <windows.h>
#include <tchar.h>

HINSTANCE g_hInstance    = NULL;
TCHAR     g_szWndTitle[] = _T("A Minimal SDK Program");
TCHAR     g_szWndClass[] = _T("MinSDK");

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMessage, WPARAM wParam, LPARAM lParam) {
    switch (uMessage) {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, uMessage, wParam, lParam);
    }

    return 0;
}

ATOM InitApplication(HINSTANCE hInstance) {
    WNDCLASSEX wcex;

    wcex.cbSize        = sizeof(WNDCLASSEX);
    wcex.style         = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc   = WndProc;
    wcex.cbClsExtra    = 0;
    wcex.cbWndExtra    = 0;
    wcex.hInstance     = hInstance;
    wcex.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wcex.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
    wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wcex.lpszMenuName  = NULL;
    wcex.lpszClassName = g_szWndClass;

    return RegisterClassEx(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow) {
    HWND hWnd = CreateWindow(
```

```
        g_szWndClass,
        g_szWndTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL);

    if (! hWnd)
        return FALSE;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    g_hInstance = hInstance;

    return TRUE;
}

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR pCmdLine, int nCmdShow) {
    if (!hPrevInstance)
        if (!InitApplication(hInstance))
            return FALSE;

    if (!InitInstance(hInstance, nCmdShow))
        return FALSE;

    MSG message;

    while (GetMessage(&message, NULL, 0, 0)) {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    return message.wParam;
}
```

- C++语言

```
// minimfc.cpp

#include <afxwin.h>

class CMinApp : public CWinApp {
public:
    BOOL InitInstance(void) {
        CFrameWnd* pFrameWnd = new CFrameWnd;

        pFrameWnd -> Create(0, _T("A Minimal MFC Program"));
        pFrameWnd -> ShowWindow(SW_SHOWNORMAL);
        pFrameWnd -> UpdateWindow();
```

```
        m_pMainWnd = pFrameWnd;

        return TRUE;
    }
};

CMinApp MinApp;
```

1.2.2 类型检查更严格

- C语言

```c
/* ccheck.c */

void func(unsigned char* puc) {
}

int main(void) {
    char str[] = "hello";

    func(str); /* 参数类型不匹配也能通过编译 */

    return 0;
}
```

- C++语言

```cpp
// cppcheck.cpp

void func(unsigned char* puc) {
}

int main(void) {
    char str[] = "hello";

    func(str); // 参数类型不匹配无法通过编译

    return 0;
}
```

1.2.3 支持真正意义上的枚举类型

- C语言

```c
/* cenum.c */

/* 在C语言中，枚举可以直接按整数处理 */

#include <stdio.h>

typedef enum tag_LineStyle {
    ELINESTYLE_SOLID,
    ELINESTYLE_DASH,
```

```
    ELINESTYLE_DOT,
    ELINESTYLE_DASHDOT,
    ELINESTYLE_DASHDOTDOT,
    ELINESTYLE_NULL
}   ELINESTYLE;

void drawLine(ELINESTYLE lineStyle, double startX, double startY, double endX, double endY) {
    /* ... */
}

int main(void) {
    drawLine(0, 0.0, 0.0, 100.0, 100.0); /* 以整型实参对应枚举型形参 */

    return 0;
}
```

- C++语言

```
// cppenum.cpp

// 在C++语言中，枚举是一个独立的数据类型，无法与整型互换

#include <stdio.h>

typedef enum tag_LineStyle {
    ELINESTYLE_SOLID,
    ELINESTYLE_DASH,
    ELINESTYLE_DOT,
    ELINESTYLE_DASHDOT,
    ELINESTYLE_DASHDOTDOT,
    ELINESTYLE_NULL
}   ELINESTYLE;

void drawLine(ELINESTYLE eLineStyle, double startX, double startY, double endX, double endY) {
    // ...
}

int main(void) {
    drawLine(0, 0.0, 0.0, 100.0, 100.0); // 将整型实参传给枚举形参，编译时错误，非法转换

    return 0;
}
```

1.2.4 参数匹配更精确

- 最优匹配原则

```
// optimal.cpp

// C++语言函数调用时的最优匹配原则

#include <iostream>
```

```
6
7  using namespace std;
8
9  void func(int n) {
10     cout << "func(int) invoked" << endl;
11 }
12
13 void func(char c) {
14     cout << "func(char) invoked" << endl;
15 }
16
17 int main(void) {
18     func('c'); // 匹配到func(char)版本
19
20     return 0;
21 }
```

- 升级优先原则

```
1  // priupgrade.cpp
2
3  // C++语言函数调用时的升级优先原则
4
5  #include <iostream>
6
7  using namespace std;
8
9  void func(int n) {
10     cout << "func(int) invoked" << endl;
11 }
12
13 void func(char c) {
14     cout << "func(char) invoked" << endl;
15 }
16
17 int main(void) {
18     func(true); // 匹配到func(int)版本
19
20     return 0;
21 }
```

- 同名隐藏原则

```
1  // hide.cpp
2
3  // C++语言函数调用时的同名隐藏原则
4
5  #include <iostream>
6
7  using namespace std;
8
9  void func(int n) {
10     cout << "func(int) invoked" << endl;
11 }
12
```

```cpp
void func(char c) {
    cout << "func(char) invoked" << endl;
}

int main(void) {
    func(10); // 匹配到func(int)版本

    void func(char c); // 后声明的标识符隐藏先声明的同名标识符

    func(10); // 匹配到func(char)版本

    return 0;
}
```

- 等差歧义原则

```cpp
// amb.cpp

// C++语言函数调用时的等差歧义原则

#include <iostream>

using namespace std;

void func(int n) {
    cout << "func(int) invoked" << endl;
}

void func(char c) {
    cout << "func(char) invoked" << endl;
}

int main(void){
    func(0.614); // 编译时错误，歧义

    return 0;
}
```

1.2.5 禁止使用隐式声明

- C语言

```c
/* cimpdcl.c */

/* C语言中返回值类型为int的函数可以不先声明而直接调用，谓之隐式声明 */

#include <stdio.h>

int main(void) {
    int n = func(); /* 返回值类型为int的函数可以未经声明而直接调用 */

    printf("n=%d\n", n);

    return 0;
}
```

```
14
15 int func(void) {
16     return 1000;
17 }
```

- C++语言

```
1  // cppimpdcl.cpp
2
3  // C++语言不允许隐式声明
4
5  #include <iostream>
6
7  using namespace std;
8
9  int main(void) {
10     int n = func(); // 编译时错误，未声明
11
12     cout << "n=" << n << endl;
13
14     return 0;
15 }
16
17 int func(void) {
18     return 1000;
19 }
```

1.3 更丰富的语法特性

1.3.1 名字空间

1.3.1.1 为什么要用名字空间?

- 通过名字空间可以对代码进行逻辑划分

```
1  // modules.cpp
2
3  // 用名字空间对代码进行逻辑划分
4
5  // 压缩解压缩模块
6  namespace compress {
7      void zip(void) {
8          // ...
9      }
10
11     void unzip(void) {
12         // ...
13     }
14 }
15
16 // 加密解密模块
17 namespace encrypt {
18     void des(void) {
19         // ...
20     }
```

```
21
22      void rsa(void) {
23          // ...
24      }
25  }
26
27  // 网络通信模块
28  namespace network {
29      void send(void) {
30          // ...
31      }
32
33      void recv(void) {
34          // ...
35      }
36  }
37
38  int main(void) {
39      compress::zip();
40      compress::unzip();
41
42      encrypt::des();
43      encrypt::rsa();
44
45      network::send();
46      network::recv();
47
48      return 0;
49  }
```

- 名字空间有助于避免名字冲突

```
1   // nameconf.cpp
2
3   // 用名字空间防止名字冲突
4
5   #include <iostream>
6
7   using namespace std;
8
9   // 工行支付接口
10  namespace icbc {
11      void pay(double money){
12          cout << "icbc::pay(" << money << ") invoked" << endl;
13      }
14  }
15
16  // 建行支付接口
17  namespace ccb {
18      void pay(double money) {
19          cout << "ccb::pay(" << money << ") invoked" << endl;
20      }
21  }
22
23  int main(void) {
24      icbc::pay(100.00);
```

```
    ccb::pay(200.00);

    return 0;
}
```

1.3.1.2 怎样定义名字空间?

- 基本语法

```
namespace 名字空间名 {
    名字空间成员
}
```

- 在已有名字空间中添加成员

```
// append.cpp

// 在已有名字空间中添加成员

// 定义rip名字空间
namespace rip {
    void drawText(double x, double y, const char* text) {
        // ...
    }

    void drawRect(double left, double top, double right, double bottom)
{
        // ...
    }
}

// 在rip名字空间中添加drawCircle成员
namespace rip {
    void drawCircle(double x, double y, double radius) {
        // ...
    }
}

int main(void) {
    rip::drawText(100.0, 200.0, "hello world");
    rip::drawRect(300.0, 400.0, 500.0, 600.0);
    rip::drawCircle(700.0, 800.0, 900.0);

    return 0;
}
```

- 名字空间的声明与定义可以分开

```
// divided.cpp

// 名字空间的声明与定义可以分开

#include <iostream>

```

```cpp
using namespace std;

namespace docbase {
    extern unsigned int ndocs; // 在名字空间中声明变量

    class Document { // 在名字空间中声明类
    public:
        Document(void);
        ~Document(void);

        void open(const char* path);
        void close(void);
        void save(void);
        void saveAs(const char* path);
    };

    Document* createDocument(void); // 在名字空间中声明函数
}

// 定义名字空间中的变量、类和函数

unsigned int docbase::ndocs = 0;

docbase::Document::Document(void) {
    // ...
}

docbase::Document::~Document(void) {
    // ...
}

void docbase::Document::open(const char* path) {
    // ...
}

void docbase::Document::close(void) {
    // ...
}

void docbase::Document::save(void) {
    // ...
}

void docbase::Document::saveAs(const char* path) {
    // ...
}

docbase::Document* docbase::createDocument(void) {
    Document* doc = new Document;

    ndocs++;

    return doc;
}

int main(void) {
```

```
    cout << docbase::ndocs << endl;

    docbase::Document* doc = docbase::createDocument();

    cout << docbase::ndocs << endl;

    doc->open("foo.doc");
    doc->close();

    return 0;
}
```

1.3.1.3 怎样使用名字空间?

- 作用域限定符

```
// scope.cpp

// 在名字空间外部，必须通过作用域限定符（::），引用特定名字空间的成员

#include <iostream>

using namespace std;

namespace ns {
    void foo(void) {
        cout << "ns::foo() invoked" << endl;
    }
}

int main(void) {
    ns::foo(); // ns名字空间的foo

    return 0;
}
```

- 名字空间指令

```
// diruse.cpp

// using指令表明在当前作用域中，可直接引用特定名字空间的成员

#include <iostream>

using namespace std;

namespace ns {
    void foo(void) {
        cout << "ns::foo() invoked" << endl;
    }
}

int main(void) {
    using namespace ns; // 在当前作用域中，可直接引用ns名字空间的成员

```

```
    foo();

    return 0;
}
```

- 名字空间声明

```
// impname.cpp

// using声明将特定名字空间的成员，引入当前作用域

#include <iostream>

using namespace std;

namespace ns {
    void foo(void) {
        cout << "ns::foo() invoked" << endl;
    }
}

int main(void) {
    using ns::foo; // 将ns名字空间的foo，引入当前作用域

    foo();

    return 0;
}
```

- 同时使用名字空间指令和名字空间声明

```
// dclhide.cpp

// 当前作用域中的名字，无论是直接声明的，还是通过using声明引入的，一定
// 会隐藏其可见名字空间中的同名成员，即使using指令出现在using声明之后

#include <iostream>

using namespace std;

namespace ns1 {
    void foo(void) {
        cout << "ns1::foo() invoked" << endl;
    }
}

namespace ns2 {
    void foo(void) {
        cout << "ns2::foo() invoked" << endl;
    }
}

int main(void) {
    using ns1::foo; // 将ns1名字空间的foo，引入当前作用域

```

```
    foo();

    using namespace ns2; // 在当前作用域中，可直接引用ns2名字空间的成员

    foo(); // ns2名字空间的foo，被ns1名字空间的foo隐藏

    return 0;
}
```

- 无名名字空间

```
// noname.cpp

// 无名名字空间的成员，可以直接引用，也可以通过全局作用域限定符（::）引用

#include <iostream>

using namespace std;

// 无名名字空间
namespace {
    void foo(void) {
        cout << "::foo() invoked" << endl;
    }
}

// 有名名字空间
namespace ns {
    void foo(void) {
        cout << "ns::foo() invoked" << endl;

        ::foo(); // 无名名字空间的foo
    }
}

int main(void) {
    ns::foo(); // ns名字空间的foo

    foo(); // 无名名字空间的foo

    return 0;
}
```

1.3.1.4 名字空间嵌套与名字空间别名

```
// nestalias.cpp

// 名字空间嵌套与名字空间别名

#include <iostream>

using namespace std;

namespace ns1 {
    typedef int TYPE;
```

```cpp

    namespace ns2 {
        typedef int* TYPE;

        namespace ns3 {
            typedef string TYPE;

            namespace ns4 {
                typedef string* TYPE; // 内层名字隐藏外层名字
            }
        }
    }
}

int main(void) {
    using namespace ns1;

    TYPE n = 10; // int
    cout << n << endl;

    // 嵌套名字空间逐层分解

    ns2::TYPE pn = &n; // int*
    cout << *pn << endl;

    ns2::ns3::TYPE str("hello"); // string
    cout << str << endl;

    ns2::ns3::ns4::TYPE pstr1 = &str; // string*
    cout << *pstr1 << endl;

    // 名字空间别名

    namespace NS4 = ns2::ns3::ns4;

    NS4::TYPE pstr2 = pstr1; // string*
    cout << *pstr2 << endl;

    return 0;
}
```

1.3.2 输入和输出

- C语言

```c
/* stdio.c */

/* C语言的格式化输出 */

#include <stdio.h> /* 头文件 */

int main(void) {
    char   c   = 'A';
    int    d   = 100;
    double g   = 0.224;
```

```
    char   s[] = "hello";
    int*   p   = &d;

    /* printf()函数的格式标记必须与输出数据的类型相一致 */
    printf("%c, %d, %g, %s, %p\n", c, d, g, s, p);

    return 0;
}
```

- C++语言

```
// iostream.cpp

// C++语言的格式化输出

#include <iostream> // 头文件

using namespace std;

int main(void) {
    char   c   = 'A';
    int    d   = 100;
    double g   = 0.224;
    char   s[] = "hello";
    int*   p   = &d;

    // cout为每种数据类型定义了相应的输出流插入操作符（<<），无需使用格式标记
    cout << c << ", " << d << ", " << g << ", " << s << ", " << p <<
endl;

    return 0;
}
```

1.3.3 新的数据类型

1.3.3.1 bool类型

```
// bool.cpp

// C++的bool类型

#include <iostream>

using namespace std;

int main(void) {
    bool t = true, f = false;

    cout << "Size of bool is " << sizeof(bool) << " byte(s)" << ", "
        << "true is " << t << " and false is " << f << "." << endl;

    return 0;
}
```

1.3.3.2 引用类型

- 引用即别名

```cpp
// alias.cpp

// 引用是另一个变量的别名

#include <iostream>

using namespace std;

int main(void) {
    int n = 1;
    cout << "n=" << n << endl;

    // 通过指针修改变量的值
    int* p = &n;
    *p = 2;
    cout << "n=" << n << endl;

    // 通过引用修改变量的值
    int& r = n;
    r = 3;
    cout << "n=" << n << endl;

    // 通过引用同样可以实现变量的自增(减)
    r++;
    cout << "n=" << n << endl;

    // 对引用取地址与对引用的目标取地址是一样的
    cout << "&n=" << &n << ", &r=" << &r << endl;

    // 不能声明指向引用的指针
    // int&* pr = &r; // 编译错误

    // 可以声明引用指针的引用
    int*& rp = p;
    (*rp)++;
    cout << "n=" << n << endl;

    // 引用数组的引用
    int a[] = {1, 2, 3, 4, 5};
    int (&ra)[5] = a;
    for (int i = 0; i < sizeof (ra) / sizeof (ra[0]); i++)
        cout << ra[i] << ' ';
    cout << endl;

    return 0;
}
```

- 引用必须初始化

```cpp
// init.cpp

```

```cpp
// 引用必须初始化

#include <iostream>

using namespace std;

const int c = 100;

int main(void) {
    int n;
    int& r1 = n; // 引用一经定义就必须同时初始化

    const int& r2 = 200; // 常量只能用于初始化常量引用
    cout << "r2=" << r2 << endl;

    // 试图通过常量类型转换修改常量的值可能导致不可预期的后果
    const int& r3 = c;
    const_cast<int&>(r3) = 300;
    cout << "r3=" << r3 << ", c=" << c << endl;

    return 0;
}
```

- 引用必须与有效内存相关联

```cpp
// valid.cpp

// 引用必须与有效内存相关联

#include <iostream>

using namespace std;

int main(void) {
    int* p1 = new int;
    int& r1 = *p1;
    r1 = 100;
    cout << "r1=" << r1 << endl;

    int* p2 = NULL;
    int& r2 = *p2;
    r2 = 200; // 运行错误
    cout << "r2=" << r2 << endl;

    return 0;
}
```

- 引用型参数和引用型返回值
  - 通过值、指针和引用传参

```cpp
// vprparam.cpp

// 通过值、指针和引用传参

#include <iostream>
```

```cpp

using namespace std;

// 通过值传参
void passVal(int arg) {
    ++arg; // 形参是实参的副本
}

// 通过指针传参
void passPtr(int* arg) {
    ++*arg; // 形参是实参的地址
}

// 通过引用传参
void passRef(int& arg) {
    ++arg; // 形参是实参的别名
}

int main(void) {
    int arg = 100;

    passVal(arg); // 实参不变
    cout << "passVal(): " << arg << endl;

    passPtr(&arg); // 实参被函数改变
    cout << "passPtr(): " << arg << endl;

    passRef(arg); // 实参被函数改变
    cout << "passRef(): " << arg << endl;

    return 0;
}
```

- 通过值、指针和引用返回

```cpp
// vprret.cpp

// 通过值、指针和引用返回

#include <iostream>

using namespace std;

int ret = 100;

// 返回值
int returnVal(void) {
    return ret; // 返回变量的副本
}

// 返回指针
int* returnPtr(void) {
    return &ret; // 返回变量的地址
}

// 返回引用
```

```cpp
int& returnRef(void) {
    return ret; // 返回变量的别名
}

int main(void) {
    int val = returnVal(); // 得到的是副本
    ++val;
    cout << "returnVal(): " << ret << endl;

    int* ptr = returnPtr(); // 得到的是地址
    ++*ptr;
    cout << "returnPtr(): " << ret << endl;

    int& ref = returnRef(); // 得到的是别名
    ++ref;
    cout << "returnRef(): " << ret << endl;

    return 0;
}
```

- 值到值的函数调用和引用到引用的函数调用

```cpp
// v2vr2r.cpp

// 值到值的函数调用和引用到引用的函数调用

#include <iostream>

using namespace std;

// 通过值传递参数，通过值返回
int v2v(int arg) {
    cout << "v2v formal param: " << &arg << endl; // 打印形参的地址

    return arg;
}

// 通过引用传递参数，通过引用返回
int& r2r(int& arg) {
    cout << "r2r formal param: " << &arg << endl; // 打印形参的地址

    return arg;
}

int main (void) {
    int arg = 100;
    cout << "    Actual param: " << &arg << endl; // 打印实参的地址

    int val = v2v(arg);
    cout << "v2v return value: " << &val << endl; // 打印返回值的地址

    int& ref = r2r(arg);
    cout << "r2r return value: " << &ref << endl; // 打印返回值的地址

    return 0;
```

```
}
```

- 作为函数返回值的引用，要保证在函数返回后其所引用的目标依然有效

```
// retvalid.cpp

// 作为函数返回值的引用，要保证在函数返回后其所引用的目标依然有效

#include <iostream>

using namespace std;

string str("global");

string& globalRef(void) {
    return str; // 返回全局变量的引用，安全
}

string& heapRef(void) {
    string* str = new string("heap");

    return *str; // 返回堆变量的引用，安全
}

string& staticRef (void) {
    static string str("static");

    return str; // 返回静态变量的引用，安全
}

string& actualRef(string& str) {
    return str; // 返回实参的引用，安全
}

class A {
public:
    A(void) : str("member") {}

    string& memberRef(void) {
        return str; // 返回成员变量的引用，安全
    }

private:
    string str;
};

string& localRef(void) {
    string str("local");

    return str; // 返回局部变量的引用，危险
}

int main(void) {
    cout << globalRef() << endl;

    cout << heapRef() << endl;
```

```
53
54      cout << staticRef() << endl;
55
56      string str("actual");
57      cout << actualRef(str) << endl;
58
59      A a;
60      cout << a.memberRef() << endl;
61
62      // cout << localRef() << endl; // 运行错误
63
64      return 0;
65  }
```

- 常引用型参数
  - 接受常量实参

```
1   // crparam1.cpp
2
3   // 常量引用可用于接受常量参数
4
5   #include <iostream>
6
7   using namespace std;
8
9   const double PI = 3.1415926;
10
11  double circleArea(const double& radius) {
12      // 常量引用必须被常量初始化
13      return PI * radius * radius;
14  }
15
16  int main(void) {
17      // 常量只能用于初始化常量引用
18      cout << circleArea(12.34) << endl;
19
20      return 0;
21  }
```

  - 在保证传递效率的同时防止对实参的意外修改

```
1   // crparam2.cpp
2
3   // 以常量引用作为参数可在保证传递效率的同时防止对实参的意外修改
4
5   #include <iostream>
6
7   using namespace std;
8
9   struct Student {
10      char name[256];
11      char gender[64];
12      int  age;
13      char phone[256];
14      char email[256];
```

```
    char address[1024];
};

void print(const Student& student) { // 以常量引用作为参数
    // 无法修改形参所引用实参的内容
    cout << student.name << ", " << student.gender << ", "
        << student.age++ << ", " << student.phone << ", "
        << student.email << ", " << student.address << endl; //
编译错误
}

int main(void) {
    Student student = {
        "张飞",
        "男",
        20,
        "13910110072",
        "zhangfei@tedu.cn",
        "北京市海淀区学院8号写字楼D座D509达内职业教育" };

    print(student);

    return 0;
}
```

- 引用与指针
  - 指针可以不做初始化，其目标可在初始化后随意改变（除非是指针常量），而引用必须做初始化，且一旦初始化就无法改变其所引用的目标

```
// variableptr.cpp

// 指针的目标可随意改变

#include <iostream>

using namespace std;

int main(void) {
    int i = 1;
    int* p = &i; // 初始化指针

    int j = 2;
    p = &j; // 改变指针的目标

    cout << "i=" << i << ", j=" << j << endl;

    *p = 3; // 对指针的新目标赋值

    cout << "i=" << i << ", j=" << j << endl;

    return 0;
}
```

```
// fixedref.cpp

```

```cpp
// 引用一旦初始化就无法改变其所引用的目标

#include <iostream>

using namespace std;

int main(void) {
    int i = 1;
    int& r = i; // 初始化引用

    int j = 2;
    r = j; // 对引用的目标赋值，而非改变引用的目标

    cout << "i=" << i << ", j=" << j << endl;

    r = 3; // 引用的目标一旦确定即再无变化

    cout << "i=" << i << ", j=" << j << endl;

    return 0;
}
```

- 可以定义指向指针的指针，但不能定义引用引用的引用

```cpp
int a;
int* p = &a;
int** pp = &p; // ok
```

```cpp
int a;
int& r = a;
int&& rr = r; // error
```

- 可以定义引用指针的引用，但不能定义指向引用的指针

```cpp
int a;
int* p = &a;
int*& rp = p; // ok
```

```cpp
int a;
int& r = a;
int&* pr = &r; // error
```

- 可以定义存储指针的数组，但不能定义存储引用的数组，但可以定义引用数组的引用

```cpp
int a, b, c;
int* parr[] = {&a, &b, &c}; // ok
```

```cpp
int a, b, c;
int& rarr[] = {a, b, c}; // error
```

```cpp
int arr[] = {1, 2, 3};
int (&rarr)[3] = arr; // ok
```

- 引用的本质是指针

```cpp
// refptr.cpp
//
// 引用是通过指针实现的

#include <stdio.h>

struct A {
    char& r;
};

int main(void) {
    char c;
    printf("%p\n", &c);

    A a = {c};
    char** pp = (char**)&a;
    printf("%p\n", *pp);

    return 0;
}
```

1.3.4 新的类型转换语法

1.3.4.1 通用类型转换

```cpp
// generalconv.cpp

// 通用类型转换：目标类型(源类型标识符)

#include <iostream>

using namespace std;

class A {
public:
    A(void) : m_a(100) {}

    int m_a;
};

class B {
public:
    B(void) : m_b(200), m_str("B class") {}

    void foo(void) {
        cout << "B::foo() invoked" << endl;
    }

    int m_b;
    string m_str;
};

```

```
int main(void) {
    A a;

    B* pb = (B*)(&a); // 将A*转换为B*

    pb->foo(); // 函数属于类而非对象，正常

    cout << pb->m_b << endl; // A中相同位置亦有整型成员，正常

    // cout << pb->m_str << endl; // A中无此成员，崩溃

    char* p = (char*)(pb); // 通用类型转换可在不同类型的指针间随意转换
    long l = long(pb); // 通用类型转换可在指针和整型间随意转换，不收窄即可

    // 通用类型转换不能在指针和除整型以外的其它基本类型间转换
    // char c = char(pb);
    // double f = double(pb);

    return 0;
}
```

1.3.4.2 静态类型转换

```
// staticcast.cpp

// 静态类型转换：static_cast<目标类型>(源类型标识符)

class A {};
class B : public A {};
class C {};

int main(void) {
    A* pa;
    B* pb;
    C* pc;

    // 两种不同的数据类型，只要在一个方向上能
    // 做隐式转换，就能在相反方向上做静态转换
    pa = pb;
    // pb = pa;
    pb = static_cast<B*>(pa);

    // 两种不同的数据类型，如果在任意方向上都不能
    // 做隐式转换，则在所有方向上也不能做静态转换
    // pc = pb;
    // pb = pc;
    // pc = static_cast<C*>(pb);
    // pb = static_cast<B*>(pc);

    // 任意类型的指针都能隐式转换为void*，
    // void*能静态转换为其它任意类型的指针
    void* pv = pa;
    // pa = pv;
    pa = static_cast<A*>(pv);

```

```
    return 0;
}
```

1.3.4.3 动态类型转换

```
// dynamiccast.cpp

// 动态类型转换：dynamic_cast<目标类型>(源类型标识符)

#include <iostream>

using namespace std;

class A {
public:
    virtual void what(void) {
        cout << "A class" << endl;
    }
};

class B : public A {
public:
    void what(void) {
        cout << "B class" << endl;
    }
};

class C : public B {
public:
    void what(void) {
        cout << "C class" << endl;
    }
};

int main(void) {
    B b;
    b.what();

    A* pa = &b; // 指向子类对象的基类指针
    pa->what();

    // 将基类指针动态类型转换为子类指针，若基类指针的目标
    // 确为子类或子类之子类对象，则转换成功，否则返回NULL
    B* pb = dynamic_cast<B*>(pa);
    if (pb)
        pb->what(); // 转换成功
    else
        cout << "Unable to cast A* to B*" << endl;

    C* pc = dynamic_cast<C*>(pa);
    if (pc)
        pc->what();
    else
        cout << "Unable to cast A* to C*" << endl; // 转换失败

```

```
    // 引用没有空值，故通过异常表示转换失败

    A& ra = b;
    ra.what();

    try {
        B& rb = dynamic_cast<B&>(ra);
        rb.what(); // 转换成功
    }
    catch (bad_cast& ex) {
        cout << "Unable to cast A& to B&: " << ex.what() << endl;
    }

    try {
        C& rc = dynamic_cast<C&>(ra);
        rc.what();
    }
    catch (bad_cast& ex) {
        cout << "Unable to cast A& to C&: " << ex.what() << endl; // 转换失败
    }

    return 0;
}
```

1.3.4.4 常量类型转换

```
// constcast.cpp

// 常量类型转换：const_cast<目标类型>(源类型标识符)

#include <iostream>

using namespace std;

int main(void) {
    int n = 100;

    int* p1 = &n;
    ++*p1;
    cout << n << endl;

    // 从int*到const int*可以隐式转换
    const int* p2 = p1;
    // ++*p2;

    // 从const int*到int*必须常量转换
    // int* p3 = p2;
    int* p3 = const_cast<int*>(p2);
    ++*p3;
    cout << n << endl;

    int& r1 = n;
    ++r1;
    cout << n << endl;

```

```
    // 从int&到const int&可以隐式转换
    const int& r2 = r1;
    // ++r2;

    // 从const int&到int&必须常量转换
    // int& r3 = r2;
    int& r3 = const_cast<int&>(r2);
    ++r3;
    cout << n << endl;

    return 0;
}
```

- 常量类型转换只能改变标识符的const特性，比通用类型转换更安全

```
// constsafe.cpp

// 常量类型转换只能去除标识符上的const限定，比通用类型转换更安全

#include <iostream>

using namespace std;

struct X {
    int n[4096];
    char c;
};

int main(void) {
    char c = 'A';
    cout << c << endl;

    const char* p1 = &c;

    char* p2 = const_cast<char*>(p1);
    *p2 = 'B';
    cout << c << endl;

    // X* p3 = const_cast<X*>(p1); // 编译错误

    X* p4 = (X*)p1;
    p4->c = 'C'; // 运行错误

    return 0;
}
```

- 试图通过常量类型转换修改常量的值可能导致不可预期的后果

```
// constcannot.cpp

// 试图通过常量类型转换修改常量的值可能导致不可预期的后果

#include <stdio.h>

```

```
int main(void) {
    // const int n = 123;
    const volatile int n = 123;
    printf("%p: %d\n", &n, n);

    // n = 456;
    int* p = const_cast<int*>(&n);
    *p = 456;
    printf("%p: %d\n", p, *p);
    printf("%p: %d\n", &n, n);

    return 0;
}
```

1.3.4.5 重解释类型转换

```
// reinterpretcast.cpp

// 重解释类型转换：reinterpret_cast<目标类型>(源类型标识符)

#include <stdio.h>

int main(void) {
    int n = 0x12345678;

    // 重解释类型转换可在不同类型的指针间随意转换
    char* p = reinterpret_cast<char*>(&n);
    printf("%#08x: [%#02x|%#02x|%#02x|%#02x]\n", n, p[0], p[1], p[2], p[3]);

    // 重解释类型转换可在指针和整型间随意转换，不收窄即可
    long l = reinterpret_cast<long>(p);
    printf("%#lx\n", l);
    p = reinterpret_cast<char*>(l);
    printf ("%p\n", p);

    // 重解释类型转换不能在指针和除整型以外的其它类型间转换
    // double d = reinterpret_cast<double>(p);
    // p = reinterpret_cast<char*>(d);

    return 0;
}
```

1.3.5 新的动态内存管理

```
// new.cpp

// 在堆中分配和释放内存

#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <iostream>

```

```cpp
using namespace std;

int main(void) {
    int* p1 = (int*)malloc(sizeof(int)); // 用malloc函数分配堆内存
    cout << *p1 << endl; // 堆内存默认初始化为0
    free(p1); // 用free函数释放malloc函数分配的堆内存
    // free(p1); // 不能释放已经释放过的堆内存
    p1 = NULL; // 空指针比悬空指针安全
    free(p1); // 释放空指针是空操作

    int* p2 = new int; // 用new操作符分配堆内存
    cout << *p2 << endl; // 堆内存默认初始化为0
    delete p2; // 用delete操作符释放new操作符分配的堆内存
    // delete p2; // 不能释放已经释放过的堆内存
    p2 = NULL; // 空指针比悬空指针安全
    delete p2; // 释放空指针是空操作

    int* p3 = new int(100); // 在分配堆内存的同时，将其初始化为特定值
    cout << *p3 << endl;
    delete p3;

    int* p4 = new int[5]; // 用new[]操作符在堆内存中分配数组
    for (int i = 0; i < 5; ++i)
        cout << p4[i] << ' '; // 堆内存中的数组默认初始化为0
    cout << endl;
    delete[] p4; // 用delete[]操作符释放堆内存中的数组

    int* p5 = new int[5] {100, 200, 300};
    // 在分配数组的同时，将各数组元素初始化为
    // 特定值，未指定初始值的元素被初始化为0
    for (int i = 0; i < 5; ++i)
        cout << p5[i] << ' ';
    cout << endl;
    delete[] p5;

    // 堆内存不够分配，malloc()函数会返回空指针，同时置errno
    int* p6 = (int*)malloc(0xFFFFFFFFFFFFF * sizeof(int));
    if (p6)
        free(p6);
    else
        cerr << strerror(errno) << endl;

    // 堆内存不够分配，new操作符会抛出bad_alloc异常
    try {
        int* p7 = new int[0xFFFFFFFFFFFFF];
        delete[] p7;
    }
    catch (bad_alloc& ex) {
        cerr << ex.what () << endl;
    }

    return 0;
}
```

- 当使用new运算符定义一个多维数组变量或数组对象时，它产生一个指向数组第一个元素的指针，返回的类型保持了除最左边维数外的所有维数。例如：

```
int  *p1        = new int[5];
int (*p2)[5]    = new int[4][5];
int (*p3)[4][5] = new int[3][4][5];
```

1.3.6 新的函数语法

1.3.6.1 内联

- 内联函数保持了函数的特性，却避免了函数调用的开销

```
/* dangerousmacro.c */

/* 宏的潜在风险 */

#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(3)); /* 3*3 = 9 */
    printf("%d\n", SQUARE(1+2)); /* 1+2*1+2 = 5 */

    return 0;
}
```

```
// safeinline.cpp

// 内联函数保持了函数的特性，却避免了函数调用的开销

#include <iostream>

using namespace std;

inline int square(int x) {
    return x * x;
}

int main(void) {
    cout << square(3) << endl;
    cout << square(1+2) << endl; // 内联函数也是函数，而非如宏般的文本替换

    return 0;
}
```

- inline关键字仅仅表示希望该函数被编译为内联，到底会不会成为内联由编译器决定。通常情况下大函数和递归函数都不会被处理为内联函数

```
// cannotinline.cpp

// inline关键字仅仅表示希望该函数被编译为内联，到底会不会成为内联
```

```cpp
// 由编译器决定。通常情况下大函数和递归函数都不会被处理为内联函数

#include <iostream>

using namespace std;

// 递归函数即使加了inline关键字，也不可能被编译器处理为内联函数
inline long factorial(int n) {
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}

int main(void) {
    for (int n = 0; n < 10; ++n)
        cout << n << "! = " << factorial(n) << endl;

    return 0;
}
```

- 隐式内联和显示内联
  - 隐式内联：在类声明中直接定义的函数自动被处理为内联函数

```cpp
// impinline.h

// 在类声明中直接定义的函数自动被处理为内联函数

#pragma once

class Circle {
public:
    // 内联函数
    Circle(double x, double y, double r) : x(x), y(y), r(r) {}

    // 内联函数
    double area(void) {
        return 3.1415926 * square(r);
    }

private:
    // 内联函数
    double square(double d) {
        return d * d;
    }

    double x, y, r;
};
```

```cpp
// impinline.cpp

// 在类声明中直接定义的函数自动被处理为内联函数

#include <iostream>

```

```cpp
#include "impinline.h"

using namespace std;

int main(void) {
    Circle circle(0, 0, 3);

    cout << circle.area() << endl;

    return 0;
}
```

- 显示内联：若不希望内联函数的定义出现在类声明中，则需在其声明处加inline关键字

```cpp
// expinline.h

// 若不希望内联函数的定义出现在类声明中，则需在其声明处加inline关键字

#pragma once

class Circle {
public:
    // 非内联函数
    Circle(double x, double y, double r);

    // 非内联函数
    double area(void);

private:
    // 内联函数
    inline double square(double d);

    double x, y, r;
};
```

```cpp
// expinline.cpp
//
// 若不希望内联函数的定义出现在类声明中，则需在其声明处加inline关键字

#include <iostream>

#include "expinline.h"

using namespace std;

// 非内联函数
Circle::Circle(double x, double y, double r) : x(x), y(y), r(r) {}

// 非内联函数
double Circle::area(void)
{
    return 3.1415926 * square(r);
}

// 内联函数
```

```cpp
double Circle::square (double d)
{
    return d * d;
}

int main(void) {
    Circle circle(0, 0, 3);

    cout << circle.area() << endl;

    return 0;
}
```

1.3.6.2 重载

- 同一作用域，函数名相同，参数表不同的函数构成重载关系

```cpp
// overload.cpp

// 重载函数：同一作用域，函数名相同，参数表不同

#include <iostream>

using namespace std;

void foo(int n, double d, const char* p) {
    cout << "foo(int,double,const char*) invoked" << endl;
}

// 参数的总个数不同
void foo(int n, double d) {
    cout << "foo(int,double) invoked" << endl;
}

// 对应参数的类型不同
void foo(double d, int n, const char* p) {
    cout << "foo(double,int,const char*) invoked" << endl;
}

// 仅返回值不同，不能形成重载
// char foo (int n, double d, const char* p) {
//     return 'A';
// }

int main(void) {
    int n;
    double d;
    const char* p;

    foo(n, d, p);
    foo(n, d);
    foo(d, n, p);

    return 0;
}
```

- 重载解析：完全匹配>常量转换>升级转换>标准转换>自定义转换>省略号匹配

```cpp
// constupgrade.cpp

// 重载解析，常量转换优先于升级转换

#include <iostream>

using namespace std;

// 需要升级转换(char->int)
void foo(char* p, int n) {
    cout << "foo(char*,int) invoked" << endl;
}

// 仅需常量转换(char*->const char*)，优先
void foo(const char* p, char c) {
    cout << "foo(const char*,char) invoked" << endl;
}

int main(void) {
    char *p, c;

    foo(p, c);

    return 0;
}
```

```cpp
// upgradestd.cpp

// 重载解析，升级转换优先于标准转换

#include <iostream>

using namespace std;

// 需要标准转换(short->char)
void foo(char c) {
    cout << "foo(char) invoked" << endl;
}

// 仅需升级转换(short->int)，优先
void foo(int n) {
    cout << "foo(int) invoked" << endl;
}

// 过分的升级转换(short->long long)
void foo(long long l) {
    cout << "foo(long,long) invoked" << endl;
}

int main(void) {
    short s;

    foo(s);
```

```
28
29      return 0;
30  }
```

```
1   // ellipsis.cpp
2   //
3   // 重载解析，借助省略号所获得的匹配是最差匹配
4
5   #include <iostream>
6
7   using namespace std;
8
9   // 省略号匹配是最差匹配
10  void foo(double d, ...) {
11      cout << "foo(double,...) invoked" << endl;
12  }
13
14  // 仅需标准转换(double->int)，优先
15  void foo(int n, void* p) {
16      cout << "foo(int,void*) invoked" << endl;
17  }
18
19  int main(void) {
20      double d;
21      void* p;
22
23      foo(d, p);
24
25      return 0;
26  }
```

- 只有同一作用域中的同名函数才涉及重载问题，不同作用域中同名函数遵循标识符隐藏原则

```
1   // onescope.cpp
2
3   // 只有同一作用域中的同名函数才涉及重载问题，不同作用域中的同名函数遵循标识符隐藏原则
4
5   #include <iostream>
6
7   using namespace std;
8
9   void foo(int n) {
10      cout << "foo(int=" << n << ") invoked" << endl;
11  }
12
13  void foo(double d) {
14      cout << "foo(double=" << d << ") invoked" << endl;
15  }
16
17  int main(void) {
18      foo(10); // 通过重载解析，匹配到全局域的foo(int)
19      foo(12.3); // 通过重载解析，匹配到全局域的foo(double)
20
21      void foo(int); // 在局部域中声明foo(int)，隐藏全局域的foo(int)和
    foo(double)
```

```
22
23      foo(12.3); // 直接调用局部域的foo(int)，局部域中无重载
24
25      void foo(double); // 在局部域中声明foo(double)，与先前声明于该域的foo(int)
    构成重载
26
27      foo(12.3); // 通过重载解析，匹配到局部域的foo(double)
28
29      return 0;
30  }
```

```
1   // twoscope.cpp
2
3   // 只有同一作用域中的同名函数才涉及重载问题，不同作用域中的同名函数遵循标识符隐藏原则
4
5   #include <iostream>
6
7   using namespace std;
8
9   namespace ns1 {
10      void foo(int n) {
11          cout << "ns1::foo(int=" << n << ") invoked" << endl;
12      }
13  }
14
15  namespace ns2 {
16      void foo(double d) {
17          cout << "ns2::foo(double=" << d << ") invoked" << endl;
18      }
19  }
20
21  int main(void) {
22      using namespace ns1;
23      using namespace ns2;
24
25      foo(10); // 通过重载解析，匹配到全局域的ns1::foo(int)
26      foo(12.3); // 通过重载解析，匹配到全局域的ns2::foo(double)
27
28      using ns1::foo; // 将ns1::foo(int)引入局部域，隐藏全局域的ns1::foo(int)和
    ns2::foo(double)
29
30      foo(12.3); // 直接调用局部域的ns1::foo(int)，局部域中无重载
31
32      using ns2::foo; // 将ns2::foo(double)引入局部域，与先前引至该域的
    ns1::foo(int)构成重载
33
34      foo(12.3); // 通过重载解析，匹配到局部域的ns2::foo(double)
35
36      return 0;
37  }
```

- 函数指针的类型决定其具体指向哪个重载版本

```
1   // funcptr.cpp
2
```

```cpp
// 函数指针的类型决定其具体指向哪个重载版本

#include <iostream>

using namespace std;

void foo(int n) {
    cout << "foo(int) invoked" << endl;
}

void foo(double d) {
    cout << "foo(double) invoked" << endl;
}

int main(void) {
    void (*funcptr1)(int) = foo; // ->foo(int)
    void (*funcptr2)(double) = foo; // ->foo(double)

    funcptr1(10);
    funcptr2(12.3);

    return 0;
}
```

- 函数重载的本质是C++换名，即将函数参数表的类型信息编码到新的函数名中。以GNU C++编译器为例：
  - `foo(int,double,const char*)` → `_Z3foodiPKc`
  - `foo(int,double)` → `_Z3fooid`
  - `foo(double,int,const char*)` → `_Z3fooidPKc`
- 通过extern "C"指示C++编译器以C语言的方式处理函数接口，即不做换名，以方便C和C++代码的相互调用
  - C调C++

```cpp
// cppdecl.h

// 用C++语言编写的声明文件

#ifdef __cplusplus
extern "C" {
#endif // __cplusplus

void foo(int n, double d, const char* p);

#ifdef __cplusplus
}
#endif // __cplusplus
```

```cpp
// cppimpl.cpp

// 用C++语言编写的实现文件

#include <iostream>

#include "cppdecl.h"

using namespace std;

void foo(int n, double d, const char* p) {
    cout << "foo(int,double,const char*) invoked" << endl;
}
```

```
g++ -c cppimpl.cpp
nm cppimpl.o
...
0000000000000000 T foo
...
```

```c
/* ccall.c */

/* 用C语言编写的调用文件 */

#include "cppdecl.h"

int main(void) {
    int n;
    double d;
    const char* p;

    foo(n, d, p);

    return 0;
}
```

```
gcc ccall.c cppimpl.o -lstdc++
```

- C++调C

```c
/* cdecl.h */

/* 用C语言编写的声明文件 */

void foo(int n, double d, const char* p);
```

```c
/* cimpl.c */

/* 用C语言编写的实现文件 */

#include <stdio.h>

#include "cdecl.h"

void foo(int n, double d, const char* p) {
    printf("foo(int,double,const char*) invoked\n");
}
```

```
gcc -c cimpl.c
nm cimpl.o
...
0000000000000000 T foo
...
```

```c
// cppcall.cpp

// 用C++语言编写的调用文件

extern "C" {

#include "cdecl.h"

}

int main(void) {
    int n;
    double d;
    const char* p;

    foo(n, d, p);

    return 0;
}
```

```
g++ cppcall.cpp cimpl.o
```

- 既然extern "C"已经指示C++编译器以C语言的方式处理函数接口，即不做换名，这样的函数当然也就不可能再拥有重载版本

```c
extern "C" {

void foo(int n, double d, const char* p) {
    cout << "foo(int,double,const char*) invoked" << endl;
}
/*
void foo(int n, double d) {
    cout << "foo(int,double) invoked" << endl;
}
```

```
11 void foo(double d, int n, const char* p) {
12     cout << "foo(double,int,const char*) invoked" << endl;
13 }
14 */
15 }
```

1.3.6.3 缺省参数

- 函数的参数可以取缺省值

```
1  // defparam.cpp
2
3  // 函数的缺省(默认)参数
4
5  #include <iostream>
6
7  using namespace std;
8
9  void foo(int x, int y = 4) {
10     cout << "x=" << x << ", y=" << y << endl;
11 }
12
13 int main(void) {
14     foo(1, 2);
15     foo(3); // 未指定与形参y相对应的实参，故y取缺省值4
16
17     return 0;
18 }
```

- 函数的缺省参数只能出现在函数声明中，且必须位于参数表的右端

```
1  // defright.cpp
2
3  // 函数的缺省参数只能出现在函数声明中，且必须位于参数表的右端
4
5  #include <iostream>
6
7  using namespace std;
8
9  void foo(int x, int y, int z = 3);
10 void bar(int x, int y = 5, int z = 6);
11 void hum(int x = 7, int y = 8, int z = 9);
12 // void err(int x, int y = 6, int z); // 编译错误
13
14 int main(void) {
15     foo(1, 2);
16     bar(4);
17     hum();
18     // err(3, 9);
19
20     return 0;
21 }
22
23 void foo(int x, int y, int z /* = 3 */) {
24     cout << "foo(" << x << ',' << y << ',' << z << ") invoked" << endl;
```

```
}

void bar(int x, int y /* = 5 */, int z /* = 6 */) {
    cout << "bar(" << x << ',' << y << ',' << z << ") invoked" << endl;
}

void hum(int x /* = 7 */, int y /* = 8 */, int z /* = 9 */) {
    cout << "hum(" << x << ',' << y << ',' << z << ") invoked" << endl;
}

// void err(int x, int y /* = 6 */, int z) {
//     cout << "err(" << x << ',' << y << ',' << z << ") invoked" << endl;
// }
```

- 缺省参数可能引发重载歧义

```
// defamb.cpp

// 缺省参数可能引发重载歧义

#include <iostream>

using namespace std;

void foo(int x, int y = 4) {
    cout << "x=" << x << ", y=" << y << endl;
}

void foo(int x) {
    cout << "x=" << x << endl;
}

int main (int argc, char* argv[]) {
    foo(1, 2);
    foo(3); // 编译错误，既可以匹配到foo(int,int)，y取缺省值，也可以匹配到foo(int)，歧义

    return 0;
}
```

1.3.6.4 哑元

- 借助哑元参数保证函数调用的向后兼容

```
// forbkwd.cpp

// 借助哑元参数保证函数调用的向后兼容
/*
// Ver. 1.0
void func (bool b) {
    if (b) {
        // ...
    }
    else {
```

```
        // ...
    }
}
*/
// Ver. 2.0
void func (bool) { // 升级后的版本已经不再需要参数
    // ...
}

int main (int argc, char* argv[]) {
    func (true); // 哑元参数使函数调用语句无需修改

    return 0;
}
```

- 借助哑元参数构造函数的重载版本

```
// forover.cpp

// 借助哑元参数构造函数的重载版本

#include <iostream>

using namespace std;

void foo(int n) {
    cout << "foo(int) invoked" << endl;
}

void foo(int n, int) { // 哑元参数仅为与foo(int)构成重载
    cout << "foo(int,int) invoked" << endl;
}

int main(void) {
    foo(1); // 匹配到foo(int)
    foo(2, 3); // 匹配到foo(int,int)

    return 0;
}
```

1.4 来自C++社区的建议

1.4.1 数据声明与函数可执行代码相混合

- C语言

```
/* cdeclare.c */

/* 传统意义上的C语言要求作用域内的任何声明都必须位于该作用域的开始处 */

#include <stdlib.h>

size_t getFilesize(const char* filepath) {
    size_t filesize;

```

```c
    /* ... */

    return filesize;
}

int main(void) {
    /* 对局部变量的声明必须位于函数中任何可执行语句之前 */
    size_t filesize;
    unsigned char* filedata;
    int i;

    /* ... */

    filesize = getFilesize("foo.dat");

    if (filesize > 0) {
        filedata = (unsigned char*)malloc(filesize);

        /* ... */

        free(filedata);
    }

    for (i = 0; i < 10; ++i) {
        /* ... */
    }

    return 0;
}
```

- C++语言

```cpp
// cppdeclare.cpp

// C++语言允许数据声明与函数可执行代码相混合

#include <stdlib.h>

size_t getFilesize(const char* filepath) {
    size_t filesize;

    // ...

    return filesize;
}

int main(void) {
    // ...

    // 变量的声明、定义和初始化在同一条语句中完成
    size_t filesize = getFilesize("foo.dat");

    if (filesize > 0) {
        // 在语句块内部定义只在该块中使用的变量
        unsigned char* filedata = new unsigned char[filesize];

```

```
        // ...

        delete[] filedata; // 用new/delete取代malloc/free
    }

    for (int i = 0; i < 0; ++i) { // 在循环语句内部定义循环控制变量
        // ...
    }

    return 0;
}
```

1.4.2 用const、enum、inline和namespace取代宏和条件编译

- C语言

```
/* cmacro.c */

/* C语言程序中宏和条件编译的使用 */

#include <stdio.h>

#define PI 3.1415926 // 常量宏

#ifdef TARENA_GDI // 条件编译宏
    #define SHAPE_RECT      0 // 唯一标识宏
    #define SHAPE_ROUNDRECT 1
    #define SHAPE_CIRCLE    2
    #define SHAPE_ELLIPSE   3
#elif defined (MOZILA_GDI)
    #define SHAPE_RECT      10
    #define SHAPE_ROUNDRECT 11
    #define SHAPE_CIRCLE    12
    #define SHAPE_ELLIPSE   13
#endif

#define max(a, b) (((a) > (b)) ? (a) : (b)) // 参数宏

int main(void) {
    printf("PI=%g\n", PI);

    printf("SHAPE_RECT=%d\n", SHAPE_RECT);
    printf("SHAPE_ROUNDRECT=%d\n", SHAPE_ROUNDRECT);
    printf("SHAPE_CIRCLE=%d\n", SHAPE_CIRCLE);
    printf("SHAPE_ELLIPSE=%d\n", SHAPE_ELLIPSE);

    printf("max(21,37)=%d\n", max(21, 37));

    return 0;
}
```

- C++语言

```
// cppconst.cpp
```

```cpp

// C++语言程序用const取代常量宏，用enum取代唯一标识宏，用
// inline取代参数宏，用namespace取代条件编译宏解决名字冲突

#include <iostream>

using namespace std;

const double PI = 3.1415926; // 用const取代常量宏

namespace tarena_gdi { // 用namespace取代条件编译宏解决名字冲突
    enum Shape { // 用enum取代唯一标识宏
        SHAPE_RECT,
        SHAPE_ROUNDRECT,
        SHAPE_CIRCLE,
        SHAPE_ELLIPSE
    };
}

namespace mozila_gdi {
    enum Shape {
        SHAPE_RECT = 10,
        SHAPE_ROUNDRECT,
        SHAPE_CIRCLE,
        SHAPE_ELLIPSE
    };
}

inline int max(int a, int b) { // 用inline取代参数宏
    return a > b ? a : b;
}

int main(void) {
    using namespace tarena_gdi;

    cout << "PI=" << PI << endl;

    cout << "SHAPE_RECT=" << SHAPE_RECT << endl;
    cout << "SHAPE_ROUNDRECT=" << SHAPE_ROUNDRECT << endl;
    cout << "SHAPE_CIRCLE=" << SHAPE_CIRCLE << endl;
    cout << "SHAPE_ELLIPSE=" << SHAPE_ELLIPSE << endl;

    cout << "max(21,37)=" << max(21, 37) << endl;

    return 0;
}
```

1.4.3 用string和STL容器取代低级数组

- C语言

```cpp
/* cstring.c */

/* 复杂而危险的C风格字符串 */

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    /* 缺乏独立的字符串类型，用字符数组表示字符串 */
    char firstName[] = "Bjarne";
    char familyName[] = "Stroustup";

    char* fullName = malloc(strlen(firstName) + strlen(familyName) + 2);
    /* 缓冲区分配不足极易导致溢出 */

    strcpy(fullName, firstName); /* 危险的strcpy */
    strcat(fullName, " "); /* 危险的strcat */
    strcat(fullName, familyName); /* 危险的strcat */

    printf("%s\n", fullName); /* 需要正确使用格式化占位符 */

    free(fullName); /* 自己维护缓冲区的分配与释放 */

    return 0;
}
```

- C++语言

```cpp
// cppstring.cpp

// 简洁而安全的C++字符串

#include <iostream>

using namespace std;

int main(void) {
    // std提供了独立的字符串类型，自行维护缓冲区，对程序员透明
    string firstName = "Bjarne";
    string familyName = "Stroustup";

    // 重载加号运算符，代码更简洁，缓冲区自动扩展，使用更安全
    string fullName = firstName + " " + familyName;

    // 输入输出流直接支持字符串类型
    cout << fullName << endl;

    return 0;
}
```

1.4.4 用类描述世界而非用函数处理数据

- C语言

```c
/* cfunc.c */

/* 面向过程的C语言程序，用函数处理数据 */

```

```c
/* 绘制矩形的函数 */
void drawRect(double l, double t, double r, double b) {
    /* ... */
}

/* 绘制圆形的函数 */
void drawCircle(double x, double y, double r) {
    /* ... */
}

int main(void) {
    /* 绘制矩形 */
    drawRect(100, 200, 300, 400);

    /* 绘制圆形 */
    drawCircle(500, 600, 700);

    return 0;
}
```

- C++语言

```cpp
// cppobj.cpp

// 面向对象的C++程序，用类描述世界

// 矩形类
class Rect {
public:
    Rect(double l, double t, double r, double b) : l(l), t(t), r(r), b(b) {}

    // 绘制方法
    void draw(void) {
        // ...
    }

private:
    // 属性
    double l, t, r, b;
};

// 圆形类
class Circle
{
public:
    Circle(double x, double y, double r) : x(x), y(y), r(r) {}

    // 绘制方法
    void draw(void) {
        // ...
    }

private:
    // 属性
    double x, y, r;
```

```
};

int main(void) {
    // 创建矩形对象
    Rect rect(100, 200, 300, 400);
    // 矩形绘制自己
    rect.draw ();

    // 创建圆形对象
    Circle circle(500, 600, 700);
    // 圆形绘制自己
    circle.draw ();

    return 0;
}
```