第1章 C程序设计简介

1.1 计算机系统结构概述

美籍匈牙利科学家冯·诺依曼最先提出“程序存储”的思想，并成功将其运用在计算机的设计之中，根据这一原理制造的计算机被称为冯·诺依曼结构计算机。由于他对现代计算机技术的突出贡献，因此冯·诺依曼又被称为“现代计算机之父”。现代计算机主要由控制器、运算器、存储器、输入设备、输出设备 5 部分组成，如图 1-1 所示。

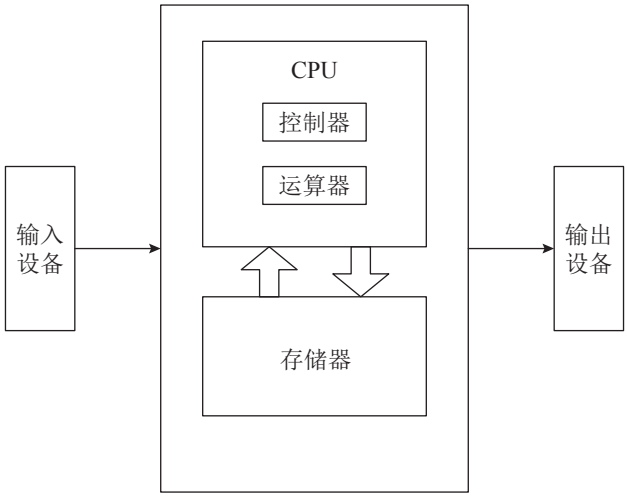

图 1-1　计算机体系结构

存储器分为内存和外存，用来存放数据和程序。内存又叫主存储器，俗称计算机中的内存条，一般采用半导体存储单元，包括随机存储器（RAM）和只读存储器（ROM），特点是存取速度快，但是容量小，价格贵；外存又叫辅存储器，常见的外存储器有硬盘、软盘、光盘、U 盘等，这类存储器一般断电后仍然能保存数据，特点是容量大，价格低，但存取速度慢。

运算器主要运行算术运算和逻辑运算，并将中间结果暂存到运算器中；控制器主要

用来控制和指挥程序与数据的输入运行，以及处理运算结果；在冯·诺依曼结构中，一般将运算器和控制器集成到一个芯片上，共同组成中央处理器（CPU），CPU 是计算机的核心。它主要有以下 4 个功能。

（1）处理指令：程序中各条指令是有严格顺序的，必须严格按程序规定的顺序执行，才能保证计算机正确工作。

（2）执行操作：一条指令的功能通常需要计算机中的部件协调完成。CPU 根据指令的功能，产生对各个部件的控制信号，从而控制部件完成指令要求的动作。

（3）控制时间：指令的执行分为不同的指令周期，每个周期所做的操作受到 CPU 的严格控制，这样计算机才能有条不紊地工作。

（4）处理数据：主要由 CPU 中的运算器执行操作，其功能主要是解释指令和对二进制数据进行算术运算或逻辑运算。

输入设备用来将程序和数据转换为机器能够识别的信息形式输入计算机里，常见的有键盘和鼠标等；输出设备可以将机器运算结果转换为人们能接受的形式或者其他系统要求的信息形式输出，如打印机输出、显示器输出等。

计算机中的指令和数据均采用二进制码表示；指令和数据以同等地位存放于存储器中；指令在存储器中按顺序存放，通常指令是按顺序执行的，特定条件下，可以根据运算结果或者设定的条件改变执行顺序；机器以运算器为中心，输入输出设备和存储器的数据传送通过运算器。

1.1.1 计算机语言发展概述

计算机擅长接受指令，但不能识别人类的语言。计算机语言是人类为了让计算机可以准确地执行指令而用于编写计算机指令的语言，也就是编写程序的语言，其本质是根据事先定义的规则编写的预定语句的集合。计算机语言总的来说分为机器语言、汇编语言和高级语言 3 大类。而这 3 种语言也恰恰是计算机语言发展历史的 3 个阶段。

计算机语言的第 1 代，称为机器语言。人类与世界第一台计算机 ENIAC 交流的语言就是机器语言。机器语言本质上是二进制码，是由程序员写出的由 0 和 1 组成的指令序列，是计算机唯一能识别的语言，人类很难理解和修改。后面的汇编语言和高级语言，都是为了让人类更好地理解计算机语言，最后给计算机执行的还是机器语言。

计算机语言发展到第 2 代，出现了汇编语言。汇编语言的实质和机器语言是相同的，都是直接对硬件操作，只不过用助记符代替了操作码，用地址符号代替地址码，例如，用 ADD 代表加法，MOV 代表数据传递等。然而计算机是不认识这些符号的，这就需要一个专门的程序，专门负责将这些符号翻译成二进制数的机器语言，这种翻译程序称为汇编程序。

计算机语言发展到第 3 代时，就进入了“面向人类”的高级语言。高级语言是一种接近人们使用习惯的程序设计语言。允许用英文写计算程序，符号和表达式也与日常的数学公式差不多。高级语言发展于 20 世纪 50—70 年代。计算机语言的发展历程如图 1-2 所示。

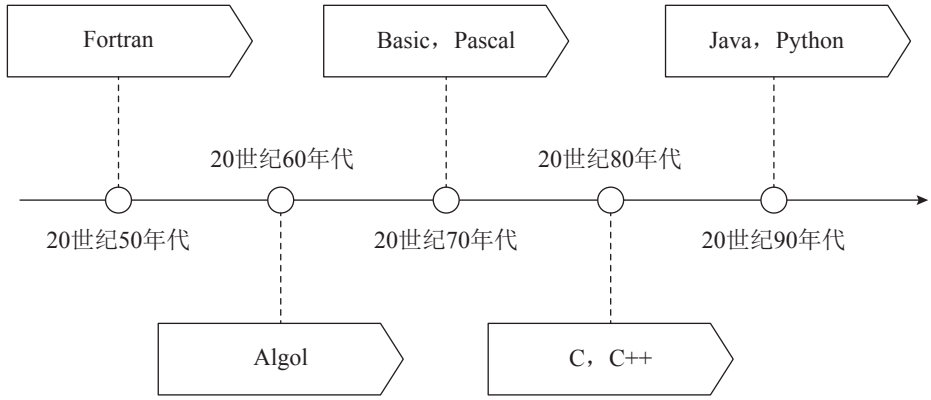

图 1-2　计算机语言的发展历程

1.1.2　C 语言起源和发展历程

20 世纪 60 年代，贝尔实验室的肯 • 汤普逊（Ken Thompson）闲来无事，想玩一个他自己编的、模拟在太阳系航行的电子游戏——*Space Travel*，他找了台空闲的机器 PDP-7，但是这台机器没有操作系统，于是肯 • 汤普逊开始用 B 语言为它开发操作系统。后来，这个操作系统被命名为 UNIX。B 语言是肯 • 汤普逊在 BCPL 语言的基础上，设计的一种更加高级的程序设计语言。1971 年，同样酷爱 *Space Travel* 的丹尼斯 • 里奇（Dennis Ritch）为了早日玩上游戏，加入了肯 • 汤普逊的开发项目，他们一起在 B 语言的基础上设计出了 C 语言。1973 年，C 语言已经足够稳定，丹尼斯 • 里奇和肯 • 汤普逊开始用 C 语言重新编写 UNIX 系统。

20 世纪 70 年代，C 语言持续发展。1977—1979 年，出现了第一本有关 C 语言的书，由丹尼斯 • 里奇和布莱恩 • 凯尼汉（Brian W. Kernighan）合作编写的 *The C Programming Language*。这本书在当时被程序员认为是 C 语言编程的标准，被称作“K&R”。

20 世纪 80 年代，C 语言已经不仅局限于 UNIX 领域。运行在不同操作系统下的多种类型的计算机都开始使用 C 语言编译器。随着 C 语言的不断发展和变化，“K&R”难以作为通用的 C 语言编写标准，影响了 C 语言的可移植性。因此，C 语言需要一个更全面更准确的统一描述。

1989 年，美国国家标准协会（ANSI）完成并通过了本国的 C 语言标准 X3.159-1989，简称 C 89 标准，有些人也把该标准称为 ANSI C。1990 年，国际标准化组织（ISO）和国际电工委员会（IEC）通过了此项标准，将其作为 ISO/IEC 9899：1990 国际标准，简称 C 90 标准。1999 年，ISO 和 IEC 通过了 ISO/IEC 9899：1999，简称 C 99 标准。2011 年，ISO 和 IEC 通过了 ISO/IEC 9899：2011，简称 C 11 标准。2018 年，ISO 和 IEC 通过了 ISO/IEC 9899：2018，简称 C 17 标准。目前最新的 C 语言标准是 C 23。

1.2 C 语言特性和应用场景

C 语言作为目前最重要、最流行的编程语言之一，对现代编程语言有很大的影响，许多现代编程语言都借鉴了 C 语言的特性。在 TIOBE 指数近 20 年的历史上，C 语言一直稳居编程语言排名前 3 名。TIOBE 编程社区指数是编程语言流行程度的指标。该指数每月更新一次，根据网络搜索引擎对含有该语言名称的查询结果的数量来计算，涵盖了网民在 Google、谷歌博客、MSN、雅虎、百度、维基百科和 YouTube 的搜索结果。

1.2.1 C 语言特性

C 语言主要有以下一些特性。

1）可移植性

C 语言的代码可移植性强，可以在不同平台上编译和运行，使得它在跨平台开发方面具备优势。这是由于 C 语言没有分裂出不兼容的多种分支，同时 C 语言的编译器规模小且容易编写。

2）高效性

C 语言提供了对硬件的直接访问能力，能进行位（bit）操作、直接内存访问等，能够实现汇编语言的大部分功能，可以直接对硬件进行操作，而且生成目标代码质量高，程序执行效率高。

3）语言简洁且功能强大

C 语言运算类型极其丰富，表达式类型多样化，支持整型、浮点型、双精度浮点型、字符型、数组类型、指针类型、结构体类型和共用体类型等各种数据类型，尤其是指针类型数据，使用十分灵活和多样化，能用来实现各种复杂的数据结构（如链表、树、栈等）的运算。

4）灵活性

C 语言可以编写从嵌入式系统到数据处理等各种应用程序，甚至可以用于制作电影的动画特效。同时，C 语言使用限制非常少，在其他语言中认定为非法的操作在 C 语言中往往是允许的。但是同时 C 语言的灵活性往往也会导致程序中的错误难以察觉。

5）代码量小

C 语言比其他许多高级语言简练，源程序短，因此输入程序时工作量少。如果要完成相同的一个功能，使用 C 语言编写的程序文件比使用其他语言编写的程序文件小很多。

6）结构化

C 语言是面向过程的程序设计语言，是完全模块化和结构化的语言。程序主要用于描述完成这个任务涉及的数据对象和具体操作规则，即设计数据结构和算法。面向过程的程序设计方法的主要思想是自顶而下规划、结构化编程和模块化设计。当需要完成一个复杂的任务时，将任务按功能分为若干基本模块，每个模块之间相对独立，对每个

模块分别进行设计，每个模块的功能实现了，整个复杂的任务就也得到了解决。面向过程的程序设计采用的流水线式的设计方法和大多数人的思维方式比较接近。模块化的编程方法，可以很好地解决一些复杂的问题，使得程序结构清晰。但是同时也存在扩展性差，代码维护困难等一些问题。

1.2.2　C 语言应用场景

C 语言的应用场景有两方面，一是系统软件开发，二是应用软件开发。而由于高效性、可移植性和灵活性等优点，C 语言主要用于系统软件的开发，如常见的三大操作系统、驱动程序（主板驱动、显卡驱动、摄像头驱动等）和数据库（SQL Server、Oracle、MySQL）等。编写应用软件虽然不是 C 语言的强项，但其依然在办公软件（WPS 等）、数学软件（MATLAB 等）、图形图像多媒体（Media Player 等）、嵌入式软件（智能手环等）和游戏开发（CS 游戏引擎等）等领域有所应用。

1.3　C 语言安全编程思考

C 语言是一种非常简单的编程语言，最初，因为 C 语言执行效率高，生成的程序几乎和汇编语言生成的程序一样快速，所以主要被用于系统性开发工作，准确地说，就是编写操作系统（如 Windows、Linux 等）和底层组件（如驱动、网络协议等）。

C 语言有非常突出的优点，当然也有一些缺点。首先，C 语言的错误往往难以察觉，这是由于 C 语言的灵活性，在其他语言中可能被发现的错误，C 编译器无法检查出来，如涉及指针的问题。

其次，C 程序可能难以理解。C 语言紧凑简洁，它的特性可以以多种方式结合起来，因此可以写出让人难以理解的代码。

最后，C 语言被认为是“不安全”的编程语言。C 语言允许使用直接内存地址进行任意指针运算，而且不进行边界检查，因此被认为是一种既缺乏与内存安全相关的特征，但又在关键系统中大量使用的编程语言。2024 年 2 月，美国白宫国家网络主任办公室（ONCD）在一份主题为“回到基础构件：通往安全软件之路”的 19 页 PDF 报告中强烈呼吁停止使用 C\C++，希望开发者抓紧使用“内存安全编程语言”。

总体来说，C 语言的优点非常多，这也是 C 语言成为最重要、最流行的编程语言之一的原因，但是同时也有难以检查错误和难以理解的问题。本书旨在引导学生在程序设计的过程中思考安全的问题，以及如何防止使用错误带来的安全隐患，规避 C 语言的缺陷。

1.3.1　安全思考

C 语言作为一种历史悠久、应用广泛的编程语言，在软件开发领域扮演着举足轻重的角色。然而，随之而来的安全性问题也是不可忽视的。C 语言的安全隐患是广泛且深刻的，深入了解这些问题对于编写安全的 C 代码至关重要。

1）内存泄漏与溢出问题

内存泄漏和溢出是C语言中最常见的安全隐患之一。内存泄漏指的是程序在动态分配内存后未正确释放该内存，导致程序运行时占用的内存不断增加，最终耗尽系统资源。缓冲区溢出是指在程序中的某个缓冲区内写入了超出其预留空间的数据，导致数据覆盖了相邻内存区域的现象。这种情况可能会造成严重的安全漏洞，甚至使得攻击者能够利用漏洞来执行恶意代码，威胁系统的安全性。

2）指针的不当使用

指针是C语言的最强大的一个工具，但也是安全隐患的来源之一。指针的不当使用可能导致指针指向未知内存区域，从而引发不可预测的行为。此外，空指针引用也是常见的错误，如果程序在引用空指针时未做适当检查，可能会导致程序崩溃。

3）语言编程上的漏洞

编程，不仅是从功能角度进行实现，还需要对编程语言的使用规则有一定理解。当对功能进行编程实现时，功能源代码会受到计算机系统本身的存储特点以及编程语言的使用规则的影响，所以实际源代码不止包括功能逻辑实现的部分，还包括更多的限制条件。例如，实现计算 (a + b + c) / d 的函数时，如果仅考虑计算逻辑的实现，那么源代码的主要部分可能如下：

```
1. int func(int result, int a, int b, int c, int d)
2. {
3.     result = (a + b + c) / d;
4. }
```

实际上这样的实现无法达到预期的目的。因为在函数被调用时，函数里的参数才会被分配空间，且在函数调用结束时，被分配的空间又会被释放，即函数中result的值无法被“带出”函数。除此之外，如果a、b、c、d分别为外部输入，当d值被恶意地输入为零时，程序会报出零为除数的非法错误；当d值被恶意地输入为极小值时，程序会大概率由于数据溢出而得不到正确结果。这些都是编程中的常见漏洞。所以编程不仅要考虑实现功能逻辑的实现，还应该考虑编程语言的规则、程序运行环境的特点，这样才能尽可能减少程序出现的问题。

1.3.2 安全实现

在考虑程序的安全实现时，主要从程序设计、程序实现和程序测试3方面去考虑。

1. 程序设计阶段须遵循的4个原则

（1）程序只实现指定的功能。

实现的程序越复杂，可能出现的漏洞和错误就越多。在程序设计阶段应该只考虑程序须实现的功能。

（2）最小权限原则。

最小权限原则是指系统的每个程序或者用户应该使用完成工作所需的最小权限工作。遵循此原则的程序在被恶意攻击者攻击时，攻击者也无法获得更多的权限做出恶意的行为。

（3）不能信任用户任何的输入。

恶意用户可以通过向程序输入恶意参数达到执行恶意行为的目的。在设计程序时，必须假设所有用户的输入不可信，来进行相应的过滤和处理，过滤恶意的输入。

（4）采用白名单机制。

很多软件都应用了黑白名单规则，被列入黑名单的用户不能通过。但是和采用黑名单机制将已知的非法操作禁止相比较，程序设计时更应该采用白名单机制，也就是只有在白名单中的用户才能通过已知的操作。

2. 程序实现阶段须考虑的两方面

（1）使用安全函数 / 安全的第三方开源工具或库函数。

在程序实现时，通常会调用现有的第三方开源工具或库函数辅助编程，减少一些固定的、可复用的功能的重复开发，但这些工具可能没有考虑安全实现，或在早期的版本存在一些漏洞。因此在实现时应该使用安全和最新的函数与工具。

（2）对代码进行静态安全检查。

代码静态分析可以由相关工具辅助完成，也须人工检阅代码。静态分析能够不断地改进代码，并更为强调程序设计时的安全思想，能够在这个步骤进一步落实对安全设计的重视。

3. 程序测试阶段的两项工作

（1）对设计的分析和评审。

在程序测试阶段需要对安全设计进行分析和评审，考虑可能出现的问题并编写测试用例。根据测试用例对程序进行测试，根据结果判断程序是否可以安全执行。

（2）安全测试。

使用常规的安全测试方法和工具等，对目标程序进行安全扫描和攻防渗透测试，寻找程序是否存在未解决的安全问题。

除了以上工作外，在安全软件开发的整个流程都有需要考虑的问题。微软提出了SDL（Security Development Lifecycle，安全开发生命周期）这一软件安全开发流程管理模式，能够更好地保证产品的安全性。

ISO 26262，DO-178B/C，FDA 等安全标准表明，编码标准可以提高程序的安全性。通过使用编码标准遵循一组规则，能够减少反复的测试和修改，降低引入错误的可能性，使代码高效且有保障地完成。常见的安全编码标准有 CWE（Common Weakness Enumeration 常见缺陷列表）、OWASP（Open Web Application Security Project 开放式 Web 应用程序安全项目）和 CERT（Computer Emergency Response Team for the Software Engineering Institute（SEI）软件工程研究所的计算机应急响应小组发布的安全编码规范）。

1.4　简单实例

1.4.1　编写一个计算机程序的过程

编写一个计算机程序的过程可以总体划分为 5 步：①问题抽象；②模块框架；③具

体实现；④程序测试；⑤调试。

第 1 步：问题抽象。在正式编写程序之前，首先需要分析待解决的问题，明确需要获取哪些信息、进行怎样的运算和判断、根据信息能够得到什么样的结果。根据分析的结果，将问题抽象成能够用流程和逻辑来表示的方案或算法。

第 2 步：模块框架。将问题抽象完成后，整理整体的框架，将相对独立的部分作为单独的一个模块实现。模块化可以对模块单独进行设计、调试和升级，容易实现模块间的互换和灵活组合，避免在同一个程序中多次重复实现同一功能模块，并且能够使得模块满足更大数量的不同问题的需要，有利于实现不同问题之间模块的通用。

第 3 步：具体实现。划分好框架和模块后，根据设计编写程序，实现目标模块和功能。在实现的过程中需要注意参数命名、代码注释和可能出现的安全问题，后面的章节将会介绍这些内容。

第 4 步：程序测试。完成编写后，需要对程序编译和执行测试。编译即将语言程序翻译成计算机能够理解的机器语言指令集，这项过程由编译器完成。编译器会在编译的过程中检查程序是否有效，如果发现错误就会报错，编译无法执行成功。需要根据提示将错误内容改正，并重新编译。编译成功后需要执行程序，测试检查程序是否按照设计的功能输出了正确的结果。如果没有得到期待的结果则说明计算机程序中存在一些错误，也就是 bug。测试的过程就是找 bug 的过程。

第 5 步：调试。在测试过程中发现 bug 后，需要查找原因和具体位置，并修复 bug，这一过程即为调试。这是保证软件正确性的必不可少的步骤。调试时可以使用模拟数据去试运行，并把输出结果与手动计算的目标结果相比较。如果存在差异，则说明存在逻辑错误。可以通过检查代码逻辑的方式修改，也可以将计算机设置成单步执行的方式，一步步跟踪程序的运行，查找出现问题的地方。找到问题后，修改程序并重新编译执行，直至程序能够正确地达到预期功能为止。

1.4.2 实例

根据 1.4.1 节的步骤，使用计算机计算两个数的和的程序步骤如下：

①问题抽象，需要使用计算机计算 10 + 109 的值，并输出结果；②模块框架，直接使用加法计算两个数的和，令它的结果等于变量 a；③具体实现，使用 C 语言实现这个功能，具体代码见下文；④程序测试，编译运行，如果编译不通过则根据提示修改代码，编译通过后则运行检查结果是否等于 119；⑤调试，如果结果不符则检查是否有逻辑错误，没有错误则不用调试。

【例 1-1】使用计算机计算 10 + 109 的值，并输出结果。

思路分析：需要两个变量存放 10 和 109，再通过加法计算，最后输出结果。

程序源代码：1-1.c

```
1. #include <stdio.h>
2. int main()
3. {
4.     /* 我的第一个 C 程序 */
5.     int a = 10;
```

```
6.    int b = 109;
7.    int c = a + b;
8.    printf("c = %d ",c);
9.    return 0;
10. }
```

该程序编译执行的结果如下：

```
c = 119
```

1.4.3　程序结构介绍

1.4.2 节中【例 1-1】是一个简单的 C 语言程序实例。一个完整的 C 语言程序由一个且只能有一个 main 函数（又称主函数，必须有）和若干其他函数结合而成（可选）。main 函数是 C 语言程序的入口，程序是从 main 函数开始执行。C 语言的函数是一段可以重复使用的代码，用来独立地完成某个功能。本节将从该实例程序出发，逐行分析程序中的代码，并介绍该实例程序的结构。

1. 编译预处理命令 #include

```
#include <stdio.h>
```

#include 是一个编译预处理命令，是在程序编译之前要处理的内容。编译预处理命令都以“#”开头，并且不用分号结尾，是 C 语言的程序语句。#include 是文件包含命令，是一个来自 C 语言的宏命令，作用是把它后面所写的那个文件的内容，完完整整地、一字不改地包含到当前的文件中来。

stdio.h 是 C 语言标准库的一个头文件。所谓头文件，其实其内容与 .c 文件中的内容是一样的，都是 C 的源代码，但头文件不用被编译。通常会把函数声明、类型和宏定义都放进头文件中，当某个 .c 源文件需要它们时，它们就可以通过一个宏命令 #include 包含进这个 .c 文件中，从而把它们的内容合并到 .c 文件中。当 .c 文件被编译时，这些被包含进去的 .h 文件的作用便发挥了。头文件 stdio.h 中定义的输入和输出函数、类型以及宏的数目几乎占整个标准库的三分之一，包含该头文件后就可以使用该文件中定义的标准输入、输出等函数了。

2. main 函数

```
int main()
```

main 函数，又称主函数，是程序执行的起点。int 指的是 main 函数的返回值的类型是整数。main 为函数名，其后的 () 括号内的内容可以为空，也可以是参数。对于有参数的 main 函数形式来说，调用这个程序的时候就需要向其传递参数。“{　}”内是代码块，一个代码块内部可以有一条或者多条语句，每条可执行代码都是以“;”英文分号结尾。

3. 注释

```
/* 我的第一个 C 程序 */
```

C 语言中的注释分为两种，行注释和块注释，它们的标识符不同，注释范围和使用方式不同。注释的内容编译器是忽略的，注释主要的作用是在代码中加一些说明和解释，这样有利于代码的阅读。

带有“//”的叫行注释，它能够注释掉所在行的位于“//”符号后面的所有内容。

“/* */”叫块注释，它能够注释掉位于“/*”和“*/”之间的所有内容。块注释是 C 语言标准的注释方法。

4. 代码内容

（1）声明和赋值。

```
int a = 10;
int b = 109;
```

int a = 10 这行代码中的 int 是 C 语言的一个关键字。关键字是程序设计语言保留下来并被赋予特定语法含义的单词或单词缩写，用来说明某一固定含义的语法概念，程序中只能使用关键字的规定作用（类似自然语言中具有特定含义的动、名词）。例如，int 在 C 语言中指的是 C 语言数据类型中的整数类型。

a 是一个标识符，所谓标识符，就是用来标识变量名、符号常量名、函数名、类型名、文件名等的有效字符序列（类似自然语言中各种事物的名字）。

该语句的功能是同时进行了声明和赋值。也就是声明在当前的代码块中有一个类型是 int、名字是 a 的变量，并给这个变量赋值为 10。本质的含义是在内存中分配一个 4 字节大小的空间，设置该内存空间的别名为 a，并把数字 10 按照整数的方式写入该内存中。

这行代码与“int a; a = 10”这两条代码等价。“=”是 C 语言的运算符号之一，表示简单赋值。

（2）运算符号。

```
int c = a + b;
```

C 语言中的符号分为 10 类：算术运算符、关系运算符、逻辑运算符、位操作运算符、赋值运算符、条件运算符、逗号运算符、指针运算符、求字节数运算符和特殊运算符。此处，“=”是赋值运算符，“+”是算术运算符，表示的是加法。“int c = a + b; ”表示将 a + b 的值赋值给变量 c。

（3）printf 函数。

```
printf("c = %d ",c);
```

printf 是 C 语言库函数，“( )”内为 main 函数传递给 printf 函数的参数，该函数的功能是向标准输出设备按规定格式输出字符串。printf() 函数的调用格式为

```
printf("< 格式化字符串 >", < 参量表 >);
```

其中，格式化字符串包括两部分内容：一部分是正常字符，这些字符可直接输出，另一部分是格式化规定字符，以“%”开始，后跟一个或几个规定字符用来确定输出内容格式。参量表是需要输出的一系列参数，其个数需要与格式化字符串中所规定的

输出参数个数一样多，各参数之间用“,”分开，且顺序一一对应，否则将会出现错误。“printf("c = %d ",c);”这行代码的意思即为按照"c = %d "格式输出内容，“%d”表示此处有一个十进制有符号整数，对应的是参数c的值。因此最终会输出“c = 119 ”。

5. return 语句

```
return 0;
```

return代表函数执行完毕，并返回一个数据，这个数据的数据类型为函数定义的数据类型。如果main定义的时候前面是int，那么return后面就需要写一个整数。

1.5 编译机制

1.5.1 编译与链接

在学习计算机还没有入门时，我们平时所说的程序，是指双击（或者在命令行窗口输入命令）后可以直接运行的程序。这样的程序在编程的角度来说，被称为可执行程序。在Windows操作系统下，通常以.exe为后缀。可执行程序内部是一系列二进制指令和数据的集合，CPU可以直接识别，但是对于大多数程序员来说，却无法记忆和使用。

可执行程序生成过程如图1-3所示。

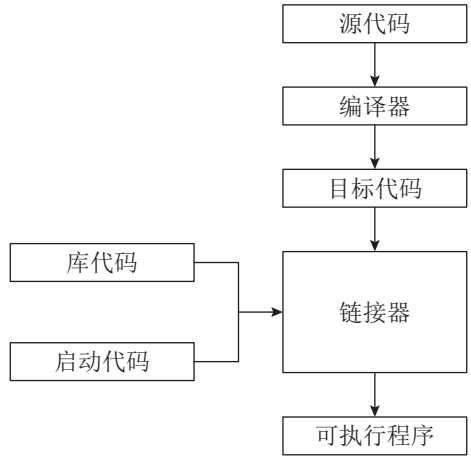

图1-3　可执行程序生成过程

其中，源代码经过编译器编译形成目标代码，然后目标代码经过链接器与所需要的库代码和启动代码链接形成可执行程序。

编译可以理解为“翻译”，类似将中文翻译成英文的过程。编译过程可以分为3个阶段：预处理阶段、编译阶段、汇编阶段，如图1-4所示。

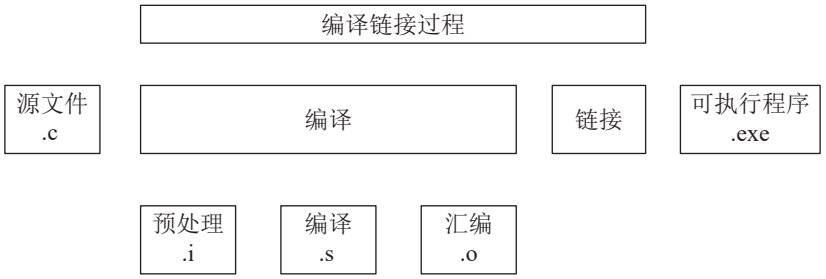

图 1-4　编译过程

1. 预处理阶段

该阶段预处理器会将 C 程序源代码文件以及相关的头文件预编译成一个 .i 文件。预处理阶段主要处理那些源代码文件中以“#”开头的预编译指令，如“#include”“#define”等，主要处理规则如下：

（1）处理“#include”预处理指令，直接将被包含的文件插入该预编译指令的位置。

（2）宏定义“#define”直接展开，并将“#define”删除。

（3）保留所有的“#pragma”编译器指令。

（4）对于注释的标记“//”“/**/”，编译器会直接忽视它们，将它们直接删除。

2. 编译阶段

编译器能够识别代码中的词汇、句子、语法，以及各种特定的格式，并将它们转换成计算机能够识别的二进制形式，这个过程称为编译。编译是一个十分复杂的过程，大致包括词法分析、语法分析、语义分析、代码优化和生成可执行文件 5 个步骤，中间涉及复杂的算法和计算机硬件架构。编译过程结束后，系统会生成一个 .s 文件。

3. 汇编阶段

汇编是指把汇编语言代码翻译成目标机器指令的过程。汇编结束后，编译所生成的 .s 文件将转换成 .o 二进制目标文件。

4. 链接阶段

链接分为静态链接和动态链接。

静态链接是由链接器在链接时将库的内容加入可执行程序中的做法。链接器是一个独立程序，功能是将一个或多个库或目标文件（先前由编译器或汇编器生成）链接到一起，生成可执行程序。静态链接是指把要调用的函数或者过程链接到可执行文件中，成为可执行文件的一部分。

动态链接所调用的函数代码和静态链接不同，并没有被复制到应用程序的可执行文件中，而是仅在其中加入了所调用函数的描述信息（往往是一些重定位信息）。仅当应用程序被装入内存开始运行时，在操作系统的管理下，才在应用程序与相应的模块之间建立链接关系。当要执行所调用模块中的函数时，根据链接产生的重定位信息，操作系统才转去执行其中相应的函数代码。

简言之，静态链接就是目标文件直接进入可执行文件。动态链接就是在程序启动后

才动态加载目标文件。静态库和应用程序编译在一起，在任何情况下都能运行。而动态库是动态链接，文件生效时才会调用。

很多代码虽然编译通过，但是链接失败就极有可能是在静态库和动态库出现了问题。缺少相关文件，就会链接报错，这个时候就要检查一下本地的链接库是不是缺损。

在 Linux 使用的 GCC 编译器把上述的预处理、编译、汇编和链接 4 个过程进行捆绑，可以只使用一次命令，就可以完成编译工作。GCC 执行过程如图 1-5 所示。

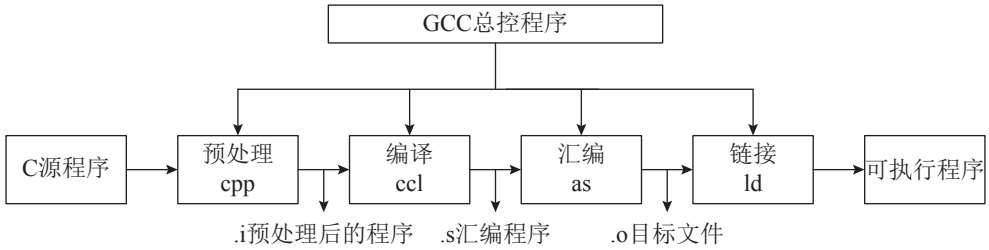

图 1-5　GCC 执行过程

GCC 命令也可以分开使用。

（1）预编译：将 .c 文件转换成 .i 文件，GCC 命令是：`gcc -E`。

（2）编译：将 .c 文件转换成 .s 文件，GCC 命令是：`gcc -S`。

（3）汇编：将 .s 文件转换成 .s 文件，GCC 命令是：`gcc -c`。

（4）链接：将 .o 文件转换成可执行程序，GCC 命令是：`gcc -o`。

对于 1.4.2 节中的【例 1-1】，我们用 GCC 如下命令对其编译：

```
gcc -S ch01_1.i -o ch01_1.s
```

得到的结果如图 1-6 所示。

```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -12(%rbp)
        movl    $109, -8(%rbp)
        movl    -12(%rbp), %edx
        movl    -8(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi endproc
```

图 1-6　.s文件内容

1.5.2 进程在内存中的布局

我们平时写的 C 代码往往会包含指针。如果把这个指针打印出来，我们会说这个指针指向某某内存地址。而这些地址在计算机的世界中并不是真实的物理地址，而是虚拟内存地址。

当一个 C 程序调入内存开始执行后，操作系统就会为这个执行的程序创建一个进程，并分配资源。每个进程都会被分配一个属于自己的内存空间，这个内存空间就是虚拟地址空间。在 32 位的操作系统下是一个 4GB 大小的地址块，这些虚拟地址通过页表映射到物理内存。对于 4GB 的地址空间（0~0xFFFFFFFF），以 Linux 平台为例，其中 1GB 必须保留给系统内核，也就是说进程自身只能拥有 3GB 的地址空间（0~0xC0000000）。这个进程的空间分布如图 1-7 所示。

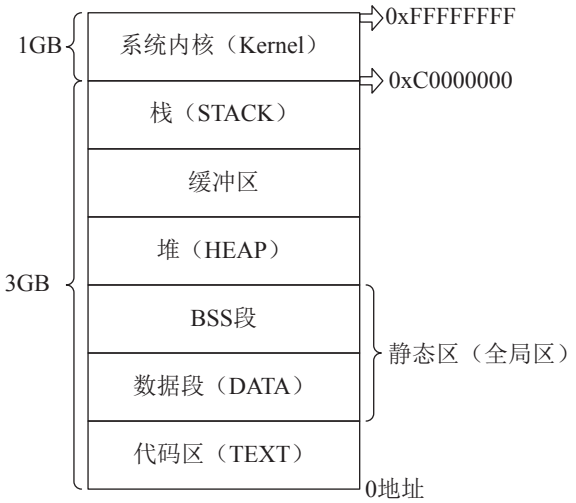

图 1-7　系统内存分布

代码区：程序（函数）代码所在，C 语言程序编译后得到的二进制代码被载入此。代码区是只读的，不允许写和其他操作，有执行权限。

数据段：用于存放程序中已初始化的全局变量和静态变量，可读可写。

BSS（Block Started by Symbol）段：通常是指用来存放程序中未初始化的全局变量和静态变量的一块内存区域。可读可写，在程序执行之前 BSS 段会自动清 0。所以，未初始化的全局变量在执行之前已经成 0 了。数据段和 BSS 段合称为静态区。

堆：自由存储区，堆区的内存分配和释放由程序员控制。在 C 语言中通常用 malloc 函数申请存储空间，用 free 函数释放所申请的空间。堆由低地址向高地址分配存储空间。

栈：由系统自动分配和释放，存储局部变量。一般情况下，人们口头上所说的堆栈，其实是指栈。栈由高地址向低地址分配存储空间。

下面给出一段代码和具体的运行结果来演示进程的空间分布。

【例 1-2】测试程序用于测试计算机上的堆栈区，并输出结果。

思路分析：定义不同的变量，依次用 printf 语句输出并查看其内存分布位置。

程序源代码：1-2.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int i1 = 10; // 静态全局区 (data 段 )
4.  int i2; // 静态全局区 (bss 段 )
5.  static int i3 = 30; // 静态全局区 (data 段 )
6.  const int i4 = 40;  // 代码区 !!!
7.  void func(int i5) // 栈区
8.  {
9.      int i6 = 60; // 栈区
10.     static int i7 = 70; // 静态全局区 (data 段 )
11.     const int i8 = 80; // 栈区 !!!
12.     char* str1 = "ABCDE"; // 代码区 ( 字符串常量 )
13.     char str2[] = "ABCDE"; // 栈区 ( 字符数组 )
14.     int* pi = (int*)malloc(sizeof(4)); // 堆区
15.     printf("&i5=%p, 这里是栈区 \n", &i5);
16.     printf("&i6=%p, 这里是栈区 \n", &i6);
17.     printf("&i7=%p, 这是静态全局区 data\n", &i7);
18.     printf("&i8=%p, 这里是栈区 \n", &i8);
19.     printf("str1=%p, 这里是代码区 \n", str1);
20.     printf("str2=%p, 这里是栈区 \n", str2);
21.     printf("pi=%p\, 这里是堆区 \n", pi);
22.     free(pi);
23. }
24. int main(void)
25. {
26.     printf("&i1=%p, 这是静态全局区 data\n", &i1);
27.     printf("&i2=%p, 这是静态全局区 bss\n", &i2);
28.     printf("&i3=%p, 这是静态全局区 data\n", &i3);
29.     printf("&i4=%p, 代码区! \n", &i4);
30.     func(50);
31.     return 0;
32. }
```

该程序执行结果如图 1-8 所示。

```
&i1=00403008, 这是静态全局区 data
&i2=004063E4, 这是静态全局区 bss
&i3=0040300C, 这是静态全局区 data
&i4=00404044, 代码区!
&i5=0061FEC0, 这里是栈区
&i6=0061FEA4, 这里是栈区
&i7=00403010, 这是静态全局区 data
&i8=0061FEA0, 这里是栈区
str1=00404048, 这里是代码区
str2=0061FE9A, 这里是栈区
pi=00B50D40, 这里是堆区
```

图 1-8　实例程序执行结果

1.6　编程规范

在自然语言里，会有词法、语法和语义存在。我们阅读语句的时候不仅要理解句子

中每个词的含义，也要注意句子的结构，从而理解语句本身的含义。

在 C 语言编程中，一样有词法、语法和语义存在。自然语言中的理解逻辑放到 C 语言编程中，一样适用，即结合上下文才能理解准确。在编译过程中，存在一个负责源码词法的部分的“词法分析器”，它帮助机器对源码进行词法分析，将源码语句解析成若干有意义的 token。同样地，也会存在语法分析器和语义分析器，前者会按照预定的语法规则把拆分的 token 组合成语句，后者会对语句的语义进行解释。

例如，下面的代码语句

```
*p = a * b;
```

通过进行词法分析，代码语句可能被解析为“*p”“=”“a”“*”“b”“;”这 6 个 token。语法分析工作会根据预定的语法规则把这些 token 划分到一个语句中，而语义分析工作会解释语句的含义，例如，这个里面出现了两次“*”符号，但是它们的意义完全不同，第一个“*”表示解引用的意思，应该和“p”结合成“*p”才是本意。第 2 个“*”是乘法符号，表示变量“a”和“b”相乘。所以即便是相同的符号，放到不同的上下文中，也会有完全不同的含义。

上述代码语句“*p = a * b;”比较短，看懂语句的含义不难。如果遇到比较长的语句，那么语句的编程方式将会影响编程人员的理解，例如：

```
if
(
x
<
y
)
x
=
a
* b;
```

上述代码片段的书写不太符合一般阅读习惯，但这不会影响编译器对语句进行分析。从开发者的角度，虽然这些语句实现了预期的功能，但不太符合自然的阅读习惯，理解语句含义的难度较大。上面的语句可以改成如下形式，既保持预期的功能，又符合阅读习惯：

```
if (x < y) x = a * b;
```

这样的表达比原来的理解起来更轻松，但还有改进的空间。作为程序开发者，不仅要写出实现预期功能的语句、符合自然阅读习惯，更重要的是，将自己对语句的理解留一个提示给其他开发者，也就是让其他开发者理解该语句的逻辑思路。该语句想做的是判断 x 和 y 的大小，如果 x 比 y 小，那么将 a * b 的值赋予 x。将这一逻辑思路化为提示，需要在语句中体现逻辑关系，该语句可以改进如下样子：

```
if (x < y)
    x = a * b;
```

这里，将语句拆分成两行，且第 2 行有缩进，一般而言，缩进在感官上有结构的内外区别，对应着“x = a * b;”该句属于在 if 逻辑语句内部。虽然如上语句体现了逻辑

关系，但在二次开发中很容易出现问题，例如，想在 if 语句中添加对 y 值的调整，那么写成

```
if (x < y)
    x = a * b;
    y = a + b;
```

看起来实现了预期功能，其实并不然，这段语句实际上等价于

```
if (x < y)
{
    x = a * b;
}
y = a + b;
```

也就是无论 if 内的语句有没有执行，“y = a + b;”一定会执行，这就和预期有出入。所以，在实际开发中，不管 if 内部的语句有多少，都应该加上括号，强调 if 语句的作用范围，避免不必要的 bug。

个人开发者当然可以按照自己的兴趣习惯，采用极具个人特色的编程方式，但是在团队开发中，每个人的这种特色编程，很可能成为团队开发的阻碍，例如，某个开发者的个人特色会影响其他开发者对合作程序源代码的理解。为此，在编程开发中，需要达成一些共识，例如，赋予一些变量名固定的含义、实现某类功能的函数的命名准则等。越是大规模的团队开发，这些共识越是能降低每个开发者的理解成本和接手难度。

良好的编程规范，不仅能降低理解难度，还能有效防止一些 bug 的出现，从而提高程序的安全性。如下的例子相信是绝大多数开发者的第一个 C 程序。

【例 1-3】使用计算机打印“hello world！”。

程序源代码：1-3.c

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     printf("hello world!");
6. }
```

这段代码虽然能够编译通过并运行成功，功能上也看不出错误，起码从计算机的角度看，它是正确的。实际上，它隐含了一个不易察觉的错误，即在函数中，没有给出返回值。如果函数没有显式声明返回值，则会根据函数返回类型，选择默认值返回。这是一个不好的习惯。大多数程序中，函数的返回值会被作为判断函数是否成功执行的信号，一般而言，返回 0 意味着成功，非 0 意味着执行失败。如果一个函数不显式返回开发者可控的返回值，那么编译器选择的默认返回值很可能产生意料之外的值，从而使得函数本身的执行结果无从知晓。所以，更好的写法更像是这样：

【例 1-4】使用计算机打印“hello world！”，声明返回值。

程序源代码：1-4.c

```
1. #include <stdio.h>
2.
3. int main()
```

```
4. {
5.     printf("hello world!");
6.     return 0;
7. }
```

程序编程规范在合作开发中相当重要。主流的应用程序需要经过不断的更新迭代，才能保持活力。团队也许会有人员的流动，但只要成员都遵守共同的编程规范，那么程序的开发、调试、维护都将顺利进行。

1.7 调试

调试（Debugging / Debug）又称排错，代码出现错误或者 bug，就可以用调试的方法去查找。调试的大致步骤如下：①发现程序错误的存在；②以隔离、消除等方式对错误进行定位；③确定错误产生的原因；④提出纠正错误的解决办法；⑤对程序错误予以改正，重新测试。

下述代码运行环境为 Windows 10，使用的编译器为 GNU Compiler Collection（GCC），版本为 GCC version 8.1.0（i686-posix-dwarf-rev0，Built by MinGW-W64 project），程序调试器为 GNU symbolic debugger（GDB），版本为 GNU gdb（GDB）8.1。

1.7.1 错误和警告

源代码在编译（预处理、编译、汇编）、链接、运行的各个阶段都可能会出现问题。编译器只能检查编译和链接阶段出现的问题，而可执行程序已经脱离了编译器，运行阶段出现问题编译器是无能为力的。如果代码在编译和链接阶段出现问题，编译器通常会提示有错误（error）或者警告（warning）。

错误表示程序不正确，不能正常编译、链接或运行，必须要纠正。

警告表示可能会发生错误（实际上未发生）或者代码不规范，但是程序能够正常运行，有的警告可以忽略，有的要引起注意。

错误通常分为以下 3 种。

1）编译型错误（最简单）

在编译阶段发生的错误，也称语法错误；如变量名字打错，漏掉分号等。

解决方法：直接看错误提示信息，解决问题。或者凭借经验就可以搞定。

2）链接型错误

在链接阶段，如忘记写头文件，会显示无法解析的外部符号；或者标识符写错或者不存在。

解决方法：加头文件；或者修改标识符。

3）运行时错误（最难）

在代码运行起来后的错误：如运行的结果跟预期结果不一样，发生死循环等。

解决方法：只能借助调试工具，一步一步查找。

下面将给出错误和警告的实例：

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     printf("hello world!")
6.     return 1;
7. }
```

显然，以上代码中缺少分号，这和 C 语言的语法不符合，GCC 编译器编译时会给出错误提示，如图 1-9 所示。

```
PS E:\c> gcc -c 1.c -Wall
1.c: In function 'main':
1.c:5:27: error: expected ';' before 'return'
     printf("hello world!")
                           ^
                           ;
     return 1;
     ~~~~~~
```

图 1-9　错误范例

必须在正确纠正后，代码才能正常编译、链接和运行。

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     int a[9];
6.     return 1;
7. }
```

上述这段代码在语法上符合 C 语言的规范，所以不会产生错误，但是却引发了警告，显示警告需要在命令行后加“-Wall”，警告如图 1-10 所示。

```
PS E:\c> gcc -c 1.c -Wall
1.c: In function 'main':
1.c:5:9: warning: unused variable 'a' [-Wunused-variable]
     int a[9];
         ^
```

图 1-10　警告范例

警告是说明 a 未被使用到，提醒我们是否声明了多余的变量。

程序在有警告的情况下，依然能够通过编译。在某些情况下，也能够正常运行。例如上面的警告，在声明变量未被使用的情况下，程序依然能够正常运行，但可能会浪费内存。然而有些警告会使得程序运行出错误的结果，程序虽然在继续运行，但输出的结果与预期却相差甚远，随着编程经历的丰富，对此会有更深的理解。所以，面对警告，不能视而不见，而是要弄清楚其产生的原因，如果可以，尽量减少警告，降低警告的内容对程序正确性的影响。

今后在编写代码的过程中，还会遇到许许多多的错误和警告，在遇到具体的错误和警告的时候，及时上网查阅，或自己分析代码，才能更好地进步。

1.7.2　C 语言 GDB 介绍

GNU symbolic debugger，简称 GDB，是一款命令行调试工具，主要通过输入命令

来调试程序，GDB 编译器通常以 gdb 命令的形式在终端（Shell）中使用，现在一些图形界面 IDE 的底层，也往往是 GDB 调试器。一般来说，GDB 主要帮助我们完成以下 4 方面的功能。

（1）启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。

（2）在某个指定的地方或条件下暂停程序。

（3）当程序被停住时，可以检查此时你的程序中所发生的事。

（4）在程序执行过程中修改程序中的变量或条件，将一个 bug 产生的影响修正从而测试其他 bug。

要使用 GDB 调试某个程序，该程序编译时必须加上编译选项“ -g”，否则该程序不包含调试信息。退出 GDB 可以用命令 q（quit 的缩写）或者按 Ctrl + d 键。

常用的命令如表 1-1 所示。

表 1-1　GDB 常用命令

命令名称	命令缩写	命令说明
run	r	运行一个待调试的程序
continue	c	让暂停的程序继续运行
next	n	运行到下一行
step	s	单步执行，遇到函数会进入
until	u	运行到指定行停下来
finish	fi	结束当前调用函数，回到上一层调用函数处
return	return	结束当前调用函数并返回指定值，到上一层函数调用处
jump	j	将当前程序执行流跳转到指定行或地址
print	p	打印变量或寄存器值
backtrace	bt	查看当前线程的调用堆栈
frame	f	切换到当前调用线程的指定堆栈
thread	thread	切换到指定线程
break	b	添加断点
tbreak	tb	添加临时断点
delete	d	删除断点
enable	enable	启用某个断点
disable	disable	禁用某个断点
watch	watch	监视某个变量或内存地址的值是否发生变化
list	l	显示源码
info	i	查看断点 / 线程等信息
ptype	ptype	查看变量类型
disassemble	dis	查看汇编代码
set args	set args	设置程序启动命令行参数
show args	show args	查看设置的命令行参数

GDB 的使用对于初学者而言不够直观，有些困难，所以建议在图形 IDE 上熟悉调试的方式之后，在有需要或兴趣的情况下，自行学习 GDB 调试方式。

1.7.3　C 语言中的调试

在编写代码的过程中有时会遇到逻辑错误。逻辑错误是由于设计者逻辑考虑不周到，使得代码运行和自己所想产生偏差，从而使得最后运行的结果与设想相去甚远。一般来说，逻辑错误既不会产生错误（error），也不会发生警告（warning），却使得程序最终运行结果出错。因此，我们需要使用调试（debug）方法来解决这些逻辑错误。所谓调试，就是跟踪代码的运行过程，从而发现程序的逻辑错误。在调试的过程中，我们可以监控程序的每个细节，包括变量的值，函数的调用过程，内存中数据、线程的调度等，从而发现隐藏的错误。

下面将通过具体的实例来介绍使用 GDB 工具进行调试的过程。

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      int a;
6.      int b;
7.      int c;
8.      b = 10;
9.      c = 15;
10.     a = b + c;
11. }
```

通过 GDB，可以使程序在需要的地方停止，并查看各个变量的值，从而检查变量变换是否符合要求。通过图 1-11 所示步骤，将代码进行编译并进入 GDB 调试。

```
PS E:\c> gcc  1.c -o 1.exe -g
PS E:\c> gdb 1.exe
GNU gdb (GDB) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from 1.exe...done.
(gdb)
```

图 1-11　进入 GDB 调试

通过命令 l，显示出调试中的代码，如图 1-12 所示。

通过命令 b+ 数字，可在需要的行位置打上断点，再通过命令 run 运行程序，使程序运行到断点就自动暂停运行，等待下一步指令，如图 1-13 所示。

此时，我们想要查看 a 变量的值，可以用 p 命令打印变量 a 的值，也可先得到变量 a 的地址，然后用 x 命令查询从此地址开始往后 4 字节上的值（int 型变量占用 4 字节），从而看到变量 a 在内存中存储的内容，如图 1-14 所示。

```
(gdb) l
1       #include<stdio.h>
2
3       int main()
4       {
5           int a;
6           int b;
7           int c;
8           b = 10;
9           c = 15;
10          a = b + c;
(gdb)
```

图 1-12　显示代码

```
(gdb) b 10
Breakpoint 1 at 0x4015de: file 1.c, line 10.
(gdb) run
Starting program: E:\c\1.exe
[New Thread 13856.0xacdc]
[New Thread 13856.0x36b0]

Thread 1 hit Breakpoint 1, main () at 1.c:10
10          a = b + c;
(gdb)
```

图 1-13　设置断点并运行

```
(gdb) p a
$6 = 0
(gdb) p &a
$7 = (int *) 0x61fec4
(gdb) x/4bx &a
0x61fec4:       0x00    0x00    0x00    0x00
(gdb)
```

图 1-14　查询变量值

此时，发现 a 的值为 0，并不是我们希望的 25，那是因为代码在断点处停止运行，还未运行第 10 行代码，可通过命令 n，使得代码运行下一行，再查看变量 a 的值，得到想要的结果，从而完成调试，具体如图 1-15 所示。

```
(gdb) n
11          return 1;
(gdb) p a
$8 = 25
(gdb)
```

图 1-15　再次查询变量值

对于大型程序，如果最后结果与预期不符，可以先通过分析代码逻辑来判断错误所在，如果多次检查依然无法发现问题，则需要通过运行测试数据，进入 GDB 开始调试，判断每一步的数据是否与预期相符，从而检查出程序的错误之处。

1.8 本章小结

本章对计算机系统进行概述，回顾了 C 语言的起源及其发展历程，明确 C 语言的特性和应用场景。作为目前最重要、最流行的编程语言之一，它对现代编程语言有很大的影响，许多现代编程语言都借鉴了 C 语言的特性。但是，在使用 C 语言时还需要考虑一些安全问题，在程序设计的过程中需要思考安全问题以及如何防止编码带来的安全隐患，规避 C 语言的缺陷。通过代码实例说明计算机程序结构，阐述从源代码到可执行程序的流程，并对编程规范以及代码如何调试进行说明。

习题

1. 冯 • 诺依曼结构计算机由哪些部件组成？各自的作用是什么？
2. C 语言相较其他主流编程语言有哪些优点？
3. C 语言与 C++ 语言的区别是什么？
4. 使用 C 语言编写代码时需要考虑哪些安全问题？
5. 编译和链接的区别是什么？
6. 调试的作用是什么？
7. Linux 平台下进程在内存中是如何布局的？