第 1 学期

1. C 语言

第 1 讲 编译。整数类型

1 C 语言中最简单的程序

让我们编写一个最简单的 C 语言程序，它在屏幕上显示"Hello, world"字样，除此之外什么也不做（见清单 1）。

```c
#include <stdio.h>

// This is one-line comment

/*
This is multi-line comment.
All this text is ignored by the compiler
*/

int main()
{
printf("Hello, world!\n");
return 0;
}
```

清单 1：我们的第一个程序

第 3 行展示了单行注释的例子：两个斜杠后面的所有内容直到行尾都被忽略。第 5–8 行我们看到多行注释：位于/* 和 */括号之间的所有内容，包括这些括号本身，都被忽略，不影响程序的运行。注释的目的是为自己和同事描述代码片段的功能、方式和原因。

第 10 行声明了 main 函数（子程序）。在 C 语言中，程序以一组函数的形式呈现，但其中只有一个是主函数，名为 main。程序从这里开始，并在其中调用其他函数。函数的文本包含在花括号 { 和 } 之间。第 10 行中的关键字 int 表示 main 函数返回一个整数。这个整数返回给调用 main 函数的对象，即操作系统，它将其解释为程序的退出代码：0 表示成功完成，否则为错误代码。第 13 行向操作系统返回零退出代码。请注意，执行 return 语句意味着立即终止函数的工作。因此，如果我们在第 13 行之后放置任何指令，它们将不会被执行。

一些函数从外部环境接收参数，这些参数放在函数名称后的圆括号中。在我们的例子中，main 函数不接收任何参数，所以第 10 行的括号是空的。

"Hello, world"的输出在第 12 行进行，在那里调用了 printf 函数。这个函数传递了一个参数 – 要在屏幕上显示的字符串。字符\n 表示换行。

请注意，在第 12 行调用的 printf 函数是一个库函数。它的签名（即名称、返回值类型、参数顺序和类型，但不是具体执行的指令）位于 stdio.h 文件中，我们通过第 1 行的#include 命令将其内容插入到我们的程序

中。

2 翻译器的类型

清单 1 中的程序对计算机来说是完全不可理解的，计算机只能操作零和一。为了在计算机上运行程序，首先需要将其翻译成计算机能理解的语言，即零和一的序列。为此，我们需要一个称为翻译器的程序（英语中 translator 意为翻译者）。

翻译器有两种类型：编译器和解释器。

解释器的工作方式类似于同声传译员，按顺序将源代码的每一行翻译成计算机语言。解释器逐步执行程序的方式有助于很好地定位错误。这种方法的缺点是程序代码优化程度较低，因此程序运行速度慢，占用相当多的内存并消耗大量处理器资源。

大多数翻译器属于解释器类型的编程语言被称为解释型语言。这类语言的例子包括 Python 和 Basic。

编译器的工作方式类似于文学翻译者：它一次性分析整个程序文本，在翻译每一行时考虑上下文。这使得编译器能够更好地优化机器代码。程序将运行得更快，同时占用更少的内存。另一方面，由于编译器一次性转换大块代码，所以在代码出现错误时有时更难发现，这意味着为编译器编写程序比为解释器编写程序更复杂。

大多数翻译器属于编译器类型的编程语言被称为编译型语言。这类语言包括 C、C++、Go、Rust 和 Pascal。

在使用 Mac OS X 和 Linux 操作系统时，可以使用 gcc 编译器来编译程序。假设程序文本保存在 main.c 文件中。要使用 gcc 进行编译，需要在终端中打开包含源代码的文件夹，并执行以下命令（$ 表示终端中的输入提示，不需要输入）：

```LaTeX
$ gcc main.c -o main
```

然后可以通过在终端中输入以下命令来运行程序：

```LaTeX
$ ./main
```

在 Windows 环境中，有以下选项：

1. 使用 Visual Studio 开发环境。

   从这里下载

   如何创建项目和运行代码

2. 安装 WSL（Windows Subsystem for Linux）– 一个薄虚拟化层，允许在不离开熟悉的 Windows 环境的情况下运行 Linux 应用程序。

要安装 WSL，需要在 PowerShell 中执行以下命令（以管理员身份运行）：

```LaTeX
wsl --install
```

安装完成后，运行 Ubuntu 程序 – 这就是 Linux 终端。

3 编译阶段

阶段 1. 预处理器。

在这个阶段，对程序的源文本进行预处理。预处理器在程序代码中查找所谓的预处理器指令 – 以#开头的命令，并执行指令规定的源文本操作。在清单 1 中，只有一个#include 指令，它命令预处理器将指定文件（在我们的情况下是 stdio.h 文件）的文本复制到它的位置。

此外，预处理器还从源代码中删除注释。

可以使用–E 标志查看预处理器的工作结果：

```LaTeX
$ gcc -E hello.c -o hello.i
```

阶段 2. 编译。

在这个阶段，预处理器工作结果得到的程序源代码被翻译成汇编语言 – 机器代码的人类友好表示。参见清单 2。

```LaTeX
$ gcc -S hello.i -o hello.s
```

阶段 3. 汇编。

```
.file	"main.c"
.text
.section	.rodata
.LC0:
.string	"Hello, world!"
.text
.globl	main
.type	main, @function
main:
.LFB0:
.cfi_startproc
endbr64
pushq	%rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq	%rsp, %rbp
.cfi_def_cfa_register 6
leaq	.LC0(%rip), %rax
movq	%rax, %rdi
call	puts@PLT
movl	$0, %eax
popq	%rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size	main, .-main
.ident	"GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
.section	.note.GNU-stack,"",@progbits
.section	.note.gnu.property,"a"
.align 8
.long	1f - 0f
.long	4f - 1f
.long	5
0:
.string	"GNU"
1:
.align 8
.long	0xc0000002
.long	3f - 2f
2:
.long	0x3
3:
.align 8
4:
```

在这个阶段，编译阶段得到的汇编代码被转换成机器代码（零和一）。

```
$ gcc -c hello.s -o hello.o
```

LaTeX

阶段 4. 链接。

真实项目的源代码通常占用数十万甚至数百万行，分布在几个文件中。这些文件中的每一个都经过上述三个阶段，完成后我们得到几个目标文件 file1.o, file2.o, ..., filen.o。在链接阶段，所有这些文件被连接成一个单一的可执行文件。

```LaTeX
$ gcc hello.o -o hello
```

在我们的情况下，源代码包含在一个文件中。然而，我们调用了标准库中的 printf 函数。因此，在链接阶段，hello.o 文件与库二进制文件 stdio.o 连接在一起。

4 整数数据类型

在 C 语言中，数字和任何数据一样，都存储在变量中。变量是一个命名的内存区域。每个变量都有自己的类型，在整个程序中不会改变。

C 语言中有两种主要的整数类型 – char 和 int。通过添加 signed、unsigned、long、short 等词，可以得到许多不同的整数类型。总的来说，我们得到了 char、short int、int、long int、long long int 类型，每种类型都可以是 signed 或 unsigned。

请注意，signed int 和 int 是同一类型，即 int 类型的变量总是有符号的。同时，char 类型在一个系统上可能是有符号的，而在另一个系统上可能是无符号的。因此，如果我们想在 char 中存储负数，就需要使用 signed char 类型。

整数变量声明的例子：

```C
signed int x = 2;
unsigned long long int Var123;
long int qwerty;
int x;
int y = 2;
int a, b, c = 4;
int d = y + c; // d等于6
```

最后一条命令是赋值运算符，执行后 y + c 的和将存储在分配给变量 d 的内存区域中。

变量名称区分大小写。因此，名为 cat 和 Cat 的变量是不同的。

如果声明的变量未初始化（未赋值），则其中可能包含任何内容（垃圾）。不能在算术运算中使用它。

为了知道可以在某个整数变量中存储哪些数字，需要知道它的大小。C 语言标准不规定具体的大小，但保证 char 的大小 ⩽ short int 的大小 ⩽ int 的大小 ⩽ long int 的大小 ⩽ long long int 的大小。

问题来了：如何在特定系统上知道整数类型的大小？为此，可以使用 sizeof 函数，它返回变量占用的字节数，以及 limits.h 文件中的常量。这两种方法都在清单 3 中进行了说明。典型值如图 1 所示。

请注意，sizeof(char)总是等于 1，即使该类型的变量占用不是 8 位，而是更多（根据标准，不能少于 8 位）。在这样的系统上，一个字节等于不是 8 位，而是 char 类型变量的大小（以位为单位）。其他整数变量的大小是 char 大小的倍数。

让我们试着理解为什么图 1 中的范围正是这样。为此，我们考虑一个假想的 4 位类型 half_char。让我们列出这种类型的变量可以采取的所有值，包括有符号（表 4）和无符号（表 4）。对于有符号变量，我们认为

最高位负责符号：0 对应正数，1 对应负数。请注意，在 4 位算术中，正数和与之相反的负数之和等于 0，例如 5 + (−5) = 0101 + 1011 = 10000 = 0000。

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

int main()
{
        printf("Number of bits in char: %d\n", CHAR_BIT);
        printf("Minimal value of char: %d\n", CHAR_MIN);
        printf("Maximum value of char: %d\n", CHAR_MAX);
        printf("Minimal value of signed char: %d\n", SCHAR_MIN);
        printf("Maximal value of signed char: %d\n", SCHAR_MAX);
        printf("Maximal value of unsigned char: %u\n", UCHAR_MAX);
        printf("Minimal value of short int: %d\n", SHRT_MIN);
        printf("Minimal value of short int: %d\n", SHRT_MAX);
        printf("Maximal value of unsigned short int: %u\n", USHRT_MAX);
        printf("Number of bits in int: %ld\n", sizeof(int) * CHAR_BIT);
        printf("Minimal value of int: %d\n", INT_MIN);
        printf("Maximal value of int: %d\n", INT_MAX);
        printf("Maximal value of unsigned int: %u\n", UINT_MAX);
        printf("Minimal value of long int: %ld\n", LONG_MIN);
        printf("Maximal value of long int: %ld\n", LONG_MAX);
        printf("Maximal value of unsigned long int: %lu\n", ULONG_MAX);
        return 0;
}
```

Listing 3: Определение размеров целочисленных переменных

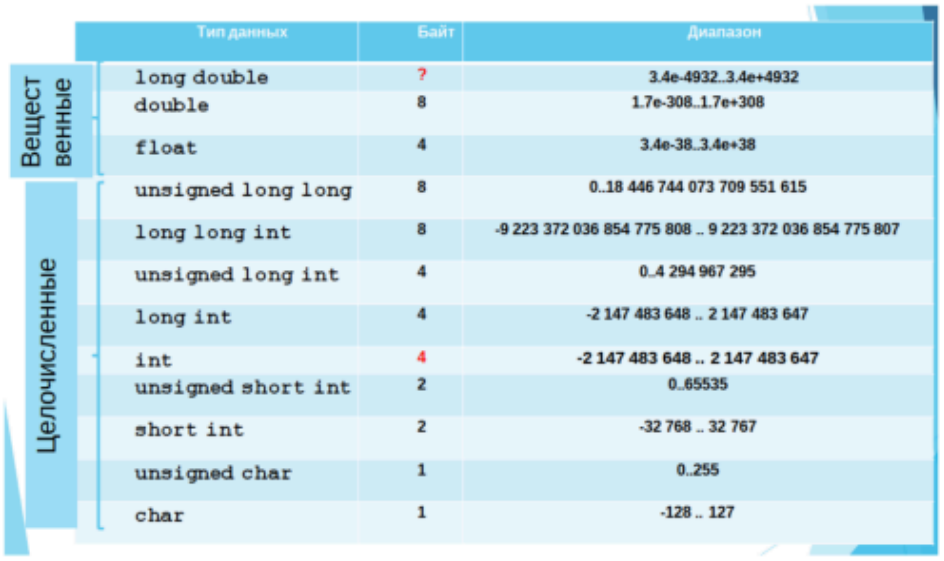


Рис. 1: Типичные размеры целочисленных типов

x_{10}	x_2
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Таблица 1: Диапазон значений беззнаковой переменной

x_{10}	x_2
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Таблица 2: Диапазон значений переменной со знаком

C 语言第二章

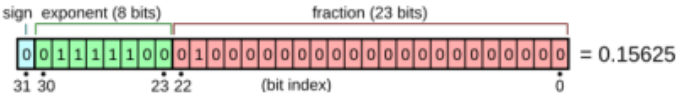


Рис. 2: Тип float

2 分数。运算符和表达式

2.1 分数的表示。实数运算的误差

任何实数都可以用指数形式表示，例如，

$$12345 = \underbrace{1.2345}_{\text{尾数}} \times 10^{\overbrace{4}^{\text{指数}}} \tag{1}$$

在二进制表示中，（1）可以写成

$$12345 = 1.\underbrace{1000000111001}_{\text{尾数}} \times 2^{\overbrace{13}^{\text{指数}}} \tag{2}$$

尾数也称为 significand 或 fraction。指数也称为幂。

在 C 语言中，分数类型表示为 float 和 double 。尽管语言标准没有规定具体的数据类型大小，但在大多数情况下， float 对应于 IEEE 754 标准中的 float32 类型（见图 2），而 double 对应于 float64 类型（见图 3）。

自然地产生两个问题：

1. float 和 double 类型可以存储的最大和最小值是多少?
2. 使用 float 和 double 类型时，分数表示的精度如何?

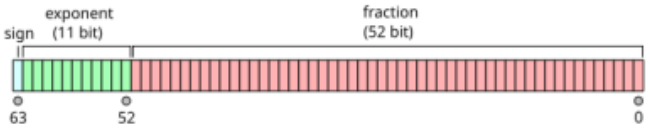


Рис. 3: Тип double

让我们以 32 位 float 为例回答这些问题。

指数（幂）占 8 位，可以编码 $2^8 = 256$ 种不同的状态，即从 0 到 255 的数字。指数始终被视为正数，并从中减去偏移量，对于 8 位指数，偏移量为 127 ($2^{8-1} - 1$)。数字 0 和 255 是保留的：255 表示无穷大（如果尾数为零）或 NaN[(1)] （如果尾数非零），而 0 允许编码次正规数 [(2)] 。因此，指数范围从 1 到 254，减去 127 的偏移量后，得到–126 到 127。考虑到尾数由 23 位编码，我们得到 2 的幂前的数字从 1 变化到 $2 - 2^{-23}$ 。因此， float32 存储的最小正值约为 $2^{-126} \approx 10^{-38}$ ，最大值约为 $(2 - 2^{-23}) \times 2^{127} \approx 10^{38}$ 。计算的相对误差为 $2^{-23} \approx 10^{-7}$ 。

类似地，对于 float64，我们可以得出相对表示误差为 $2^{-52} \approx 10^{-15}$。最小绝对值为 2^{-1022}，最大值为 $(2-2^{-52}) \times 2^{2023} \approx 10^{308}$。

清单 4 展示了使用实数时出现的误差。误差在小数点后第三位就已经出现。这是因为在处理实数时，实现的是相对误差而不是绝对误差：当处理带有 15 个零的大数时，10^{-15} 的相对误差导致单位数量级的绝对误差。

---

(1) Not a number — 非数值。

(2) 次正规数是指指数为零但尾数非零的数。对于 float32，这样的数被编码为 $(-1)^s \times 0.xxxxxx \times 2^{-126}$。次正规数平滑了最小数值和零之间的跳跃。

清单 4：执行实数运算时的舍入

```c
#include <stdio.h>
int main()
{
    double x = 123456789999999.11;
    double y = 123456789999998.1;
    printf("%lf\n", x - y); // 1.015625
    return 0;
}
```

2.2 运算符

2.2.1 算术运算，类型转换

加法、减法、乘法和除法运算符的行为总体上是可预测的。

根据 C 语言的逻辑，整数之间的运算结果是整数，包括整数除法的结果也是整数。例如，当试图将 $5/6$ 的值赋给分数变量时，答案将是 0，因为整数部分的除法结果是 0。要使结果成为分数，需要将至少一个操作数转换为分数类型 $((\text{double})5)/6$，或者简单地写成 $5.0/6$。在这两种情况下，除数本身都会转换为分数类型。

除了整数除法外，还有求余运算 $\%$。

通常，如果算术表达式中出现不同数字类型的操作数，则"较窄"的类型会转换为"较宽"的类型，操作结果也属于"较宽"的类型。

如果所有操作数都有符号，则适用以下类型转换规则：

- 如果其中一个操作数是 long double，则将另一个转换为 long double 类型。
- 否则，如果其中一个是 double，则将另一个转换为 double 类型。
- 否则，如果其中一个是 float，则将另一个转换为 float 类型。
- 否则，将 char 和 short int 转换为 int 类型。
- 然后，如果其中一个是 long，则将另一个转换为 long 类型。

如果一个操作数无符号，另一个有符号，并且变量大小相等，那么无符号类型不知为何被认为是更"宽"的，见清单 5。

清单 5：有符号和无符号数的转换

```C
#include <stdio.h>
int main()
{
    unsigned int ui = 1;
    long int li = -1;
    int i = -1;
    printf("%d\n", li < ui); // 1 - true
    printf("%d\n", i < ui);  // 0 - false
    return 0;
}
```

递增。

将变量 a 增加 b 的值：

```C#
a = a + b;
a += b;
```

增加 1：

```JavaScript
a = a + 1;
a += 1;
a++;
++a;
```

（递减类似）

后两种形式的递增/递减的区别在于，如果 ++（或--）位于右侧，则表达式的值将是变量的未递增值，见清单 6。

清单 6：递增演示

```C
#include <stdio.h>
int main()
{
    int x = 1;
    printf("%d\n", x++); // 1
    printf("%d\n", ++x); // 3
    printf("%d\n", (x++) * (++x)); // 未定义行为！
```

```
    return 0;
}
```

2.2.2 逻辑运算符

真 – 任何非零数。假 – 只有零。

与。a && b 当且仅当两个操作数都为真时为真。

或。a || b 当且仅当至少一个操作数（或两个）为真时为真。

非。!a 当且仅当 a 为假时为真。

清单 7：与运算演示

```c
#include <stdio.h>
int main()
{
    int a = 10, b = 20;
    if (a > 0 && b > 0) {
        printf("Both values are greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

清单 8：与运算演示

```c
#include <stdio.h>
int main()
{
    int a = 10, b = 20;
    if (a > 0 && b > 0) {
        printf("Both values are greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

清单 9：或运算演示

```c
```

```c
#include <stdio.h>
int main()
{
    int a = -1, b = 20;
    if (a > 0 || b > 0) {
        printf("Any one of the given value is greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

清单 10：非运算演示

```c
#include <stdio.h>
int main()
{
    int a = 10, b = 20;
    if (!(a > 0 && b > 0)) {
        printf("Both values are greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

注意。

与和或运算从左到右执行。如果与运算的第一个参数为假（或或运算的第一个参数为真），则检查第二个参数是没有意义的，因为整个表达式的值已经明确。因此，将更耗资源的表达式放在第二个（第三个等）操作数位置是有意义的。

比较运算符。 a == b, a! = b, a < b, a <= b, a > b, a >= b 。

2.2.3 位运算

按位与。运算符 & 。

按位或。运算符 | 。

按位异或（XOR）。运算符 ^。比较每两个位，如果位不同则置 1。

左移。运算符 a << n。将 a 循环左移 n 个位置。

右移。运算符 a >> n。将 a 循环右移 n 个位置。

按位非。运算符 ~a。对 a 进行按位取反。

```c
#include <stdio.h>
int main()
{
    // a = 5 (8位二进制为00000101)，b = 9 (8位二进制为00001001)
    unsigned int a = 5, b = 9;
    // 结果是 00000001
    printf("a = %u, b = %u\n", a, b);
    printf("a&b = %u\n", a & b);
    // 结果是 00001101
    printf("a|b = %u\n", a | b);
    // 结果是 00001100
    printf("a^b = %u\n", a ^ b);
    // 结果是 11111111111111111111111111111010
    // (假设32位无符号整数)
    printf("~a = %u\n", a = ~a);
    // 结果是 00010010
    printf("b<<1 = %u\n", b << 1);
    // 结果是 00000100
    printf("b>>1 = %u\n", b >> 1);
    return 0;
}
```

3 条件和循环

3.1 条件运算符

3.1.1 if-else 运算符

条件运算符允许根据条件是否满足来执行不同的指令：

```c
if (condition)
{
    // 如果条件为真则执行
}
else
{
    // 如果条件为假则执行
}
```

如果需要检查多个条件，可以使用以下结构：

```c
if (condition1)
{
    // 如果条件1为真则执行
}
else if (condition2)
{
    // 如果条件2为真则执行
}
else if (condition3)
{
    // 如果条件3为真则执行
}
else
{
    // 如果所有条件都为假则执行
}
```

如果条件不满足时不需要执行任何操作，可以省略 else：

```c
if (condition)
{
    // 如果条件为真则执行
}
```

如果只需要执行一条指令，可以省略大括号 { 和 }。

3.1.2 三元运算符

C 语言中的三元条件运算符有 3 个参数，根据第一个操作数给出的逻辑表达式的值返回第二个或第三个操作数。

C 语言中三元运算的语法如下：

```c
条件 ? 表达式1 : 表达式2;
```

如果条件成立，三元运算返回表达式 1，否则返回表达式 2。

三元运算，如同条件运算，可以嵌套。使用圆括号分隔嵌套的运算。

如果条件不满足时不需要执行任何操作，可以省略冒号和表达式 2。

3.1.3 switch 运算符

```c
switch (IntegralValue)
{
case A:
    // 如果 IntegralValue == A 则执行
    break;
case B:
    // 如果 IntegralValue == B 则执行
    break;
default:
    // 如果以上情况都不适用则执行
    break;
}
```

3.2 循环

循环允许根据某个条件是否满足重复执行相同的命令。如果只执行一条命令，可以省略大括号 { 和 }。

3.2.1 while 循环

```c
while (condition)
{
    // 当条件满足时重复执行
}
```

3.2.2 do-while 循环

```c
```

```c
do
{
    // 当条件满足时重复执行
}
while (condition)
```

与 while 循环的区别在于，无论条件是否为真， do – while 循环都会至少执行一次（因为条件是在第一次迭代后检查的）。

3.2.3 for 循环

```c
for (init; cond; incr)
{
    // 当条件满足时重复执行
}
```

for 循环的工作方式如下。在第一次迭代之前，执行 init 部分的命令。通常，这是初始化计数器变量。第二部分是条件。如果条件为真，则执行循环体。每次循环迭代后，执行第三部分 incr 中的命令。

3.3 示例

3.3.1 示例 1（计算器）

```c
#include <stdio.h>

int main()
{
    char op;
    int a, b, res;
    printf("输入运算符 (+, -, /, *) :");
    scanf("%c", &op);
    printf("输入两个操作数：");
    scanf("%d %d", &a, &b);
    if (op == '+')
        res = a + b;
    else if (op == '-')
        res = a - b;
    else if (op == '*')
        res = a * b;
    else if (op == '/')
        res = a / b;
    else
    {
        printf("您输入了错误的运算符！\n");
```

```
        return 0;
    }
    printf("结果=%d\n", res);
    return 0;
}
```

3.3.2 示例 2（二次方程根的数量）

```c
#include <stdio.h>

int main()
{
    printf("输入 a, b, c：");
    double a, b, c;
    scanf("%lf %lf %lf", &a, &b, &c);
    printf("根的数量是 %d\n",
           b*b-4*a*c > 0 ? 2 : (b*b-4*a*c == 0 ? 1 : 0));
    return 0;
}
```

3.3.3 示例 3（欧几里得算法）

```c
#include <stdio.h>

int main()
{
    int a, b;
    scanf("%d %d", &a, &b);
    while (a > 0 && b > 0)
    {
        if (a > b)
            a %= b;
        else
            b %= a;
    }
    printf("最大公约数 = %d\n", a + b);
    return 0;
}
```

3.3.4 示例 4（计算 π 值）

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...)$$

```c
#include <stdio.h>

int main()
{
    double pi = 0;
    int n;
    scanf("%d", &n);
    double factor = 1.0;
    for (int i = 0; i < n; i++, factor *= -1)
        pi += factor / (2 * i + 1);
    pi *= 4;
    printf("pi=%lf\n", pi);
    return 0;
}
```

3.3.5 示例 5（牛顿法）

考虑解方程

$$f(x) = 0$$

作为具体示例，我们取

$$\sin x - e^x = 0$$

牛顿法（或切线法）的迭代过程：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```c
#include <stdio.h>
#include <math.h>
#define EPS 1e-5

int main()
{
    double x = 1; // 初始猜测
    while (fabs(sin(x) - exp(x)) > EPS)
    {
        x -= (sin(x) - exp(x)) / (cos(x) - exp(x));
    }
    printf("x=%lf\n", x);
    printf("f=%lf\n", (sin(x) - exp(x)));
    return 0;
}
```

3.3.6 示例 6（字符串分析器）

```c
#include <stdio.h>

int main()
{
    int nwhite = 0;
    int nother = 0;
    int ndigits[10];
    for (int i = 0; i < 10; i++)
        ndigits[i] = 0;
    char ch;
    while ((ch = getchar()) != EOF)
    {
        switch (ch)
        {
        case '0': case '1': case '2':
        case '3': case '4': case '5':
        case '6': case '7': case '8': case '9':
            ndigits[ch - '0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("\n 数字\n");
    for (int i=0; i<10; i++)
        printf("%d %d\n", i, ndigits[i]);
    printf("空白字符 = %d\n", nwhite);
    printf("其他 = %d\n", nother);
    return 0;
}
```

第四章

4 内存层次结构

4.1 静态变量和全局变量

变量的作用域是程序中可以访问该变量的部分。作用域由左花括号 { 和右花括号 } 标示。在代码块内声明的所有变量只在该块内可见。

参见代码清单 11。

代码清单 11：作用域

```c
#include <stdio.h>
int main()
{
    int x = 1;
    printf("%d\n", x); // 1
    {
        int x = 2;
        printf("%d\n", x); // 2
    }
    printf("%d\n", x); // 1
    return 0;
}
```

静态变量是一种在退出作用域时不会丢失其值的变量。当再次进入相同的代码块时，静态变量不会重新初始化。与全局变量不同，静态变量只在声明它的代码块内可见。

代码清单 12 展示了静态变量与普通变量的区别。第一个循环将输出五个 1 的序列，因为每次进入第一个循环的作用域时，变量 x 都会重新初始化。第二个循环将输出从 1 到 5 的数字序列，因为静态变量只初始化一次，并且在退出代码块时其值不会丢失，即使在重新初始化时也会保持其值。

代码清单 12：静态变量

```c
#include <stdio.h>
int main()
{
    for (int i = 0; i < 5; i++)
    {
        int x = 1;
        printf("%d\n", x++);
    }
    for (int i = 0; i < 5; i++)
    {
```

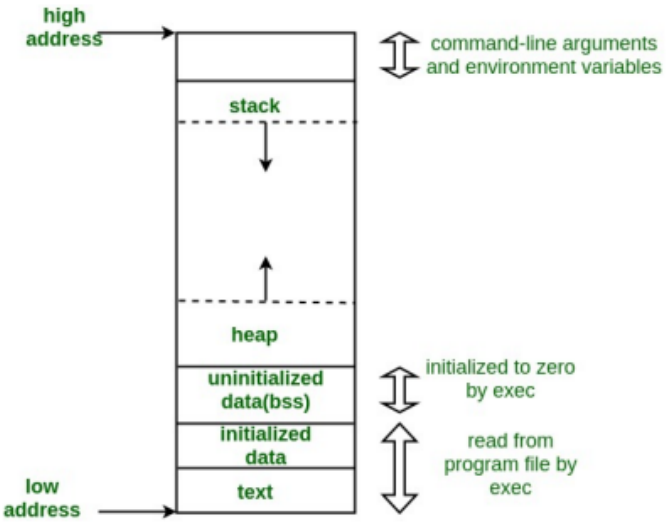
```
        static int x = 1;
        printf("%d\n", x++);
    }
    return 0;
}
```

4.2 内存层次结构

程序占用的内存可分为以下几个层次：

1. 代码段（文本段），用于存储程序指令。
2. 数据段，用于存储全局变量和静态变量。

   a) 已初始化数据段，存储由程序员初始化的静态和全局变量（在声明时被赋予初始值）。



   - 可读写子段
   - 只读子段

   b) 未初始化数据段（bss），包含未在声明时初始化的全局和静态变量。这些变量的值自动被设为 0。

3. 栈。在函数中声明的变量（例如 main 函数）。栈大小限制在几兆字节。
4. 堆。这是由程序员通过 malloc、calloc、realloc 函数显式分配的内存。

可以使用 Unix 命令 size 查看已编译程序的代码段和数据段的大小，该命令接收可执行文件名作为参数。如果您使用 Windows，请使用 WSL。

第五章

5 数组

5.1 栈上的数组

数组是同一类型变量的集合，这些变量在内存中连续存储，可以通过数组名统一访问。

数组使用示例：

代码清单 13：栈上的数组

```c
#include <stdio.h>
#define SIZE 10
int main()
{
    int a[SIZE];
    for (int i=0; i<SIZE; i++)
        scanf("%d", &a[i]);
    for (int i=0; i<SIZE; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

数组长度可以在程序运行时确定（可变长度数组 Variable Length Array）：

代码清单 14：栈上的 VLA 数组

```c
#include <stdio.h>
int main()
{
    int n;
    scanf("%d", &n);
    int a[n];
    for (int i=0; i<n; i++)
        scanf("%d", &a[i]);
    for (int i=0; i<n; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

请注意，栈的大小限制在几兆字节，因此无法在栈上创建大型数组（很可能会出现段错误 Segmentation fault）：

代码清单 15：栈太小，无法容纳这样的数组

```c
#include <stdio.h>
#define SIZE 10000000
int main()
{
    int a[SIZE];
    return 0;
}
```

5.2 全局数组

为什么要声明全局数组（在任何函数之外）？

1. 使数组不仅对 main 函数可见，也对其他函数可见。
2. 将数组放在已初始化数据段而不是栈上：后者的大小远小于前者（参见代码清单 16）。
3. 数字列表

代码清单 16：全局数组示例

```c
#include <stdio.h>
#define SIZE 10000000
int a[SIZE];
int main()
{
    return 0;
}
```

5.3 动态数组（堆上的数组）

全局数组适用于需要存储大量数据且栈空间不足的情况。但它扩展了变量的作用域（这可能是不必要的）。此外，全局数组的大小必须在编译时确定（VLA 只适用于局部数组）。

如果需要分配大型数组，且其大小在程序运行时确定，应使用堆上的数组。堆上数组的内存通过 malloc 或 calloc 函数分配，通过 free()函数释放。

代码清单 17：使用 malloc

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n;
    scanf("%d", &n);
    int *a = malloc(n * sizeof(int));
```

```
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

代码清单 18：使用 calloc

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n;
    scanf("%d", &n);
    int *a = calloc(n, sizeof(int));
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

与 malloc 不同，calloc 函数会将数组元素初始化为零。使用 malloc 分配的数组元素默认包含随机值。

代码清单 19：free()的重要性

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

int main()
{
    unsigned int n = 4000000000;
    int *a;

    while (1)
    {
        a = malloc(n * sizeof(int));
        if (a == 0)
        {
            perror("Error: ");
            break;
        }
```

```
        free(a);
    }
    return 0;
}
```

realloc 函数用于更改已分配数组的内存大小。

代码清单 20：使用 realloc()

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

int main()
{
    int n;
    scanf("%d", &n);
    int *a = calloc(n, sizeof(int));
    a = realloc(a, (n + 2) * sizeof(int));
    for (int i = 0; i < n + 2; i++)
        scanf("%d", &a[i]);
    for (int i = 0; i < n + 2; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

5.4 栈（数据结构）

"栈"一词除了表示程序的内存区域外，还指一种遵循 LIFO（后进先出）原则的数据结构。栈支持以下操作：PUSH（放入栈顶）、POP（取出并删除栈顶元素）、TOP（查看栈顶元素但不删除）。

代码清单 21 展示了栈的实现。使用了 enum 关键字，这将在后面讨论。这里只需说明该关键字定义了一个枚举类型——一种只能取花括号中列出的值的新类型。

代码清单 21：栈

第六章

6 指针与数组

6.1 指针

指针是一个包含某个内存地址的变量。

指针声明：

```c
int a = 3;
int *pa = &a;
```

& 运算符 — 获取变量的地址。

- 运算符（指针解引用）— 访问指针所存储的地址中的值。

代码清单 22：指针使用示例

```c
#include <stdio.h>
#include <stdlib.h>
int *a;
int main()
{
    int *b;
    printf("%p\n", a);
    printf("%p\n", b);
    int x = 10;
    int *c = &x;
    printf("%d\n", *c);
    x++;
    printf("%d\n", *c);
    *c = 12;
    printf("%d\n", x);
    return 0;
}
```

输出：

```ruby
(nil)
0x7813d20c9af0
10
11
12
```

如果局部指针未初始化，则它包含"垃圾"值。但是，全局声明的未初始化指针存储在 bss 段中，因此其默认值为零。

可以使用 NULL 宏（就是数字 0）来初始化指针：

```c
int *pa = NULL;
```

解引用这样的指针将导致运行时错误：

```
Segmentation fault (core dumped)
```

6.2 常量指针和指向常量的指针

考虑以下代码片段：

```c
int a = 2, b = 3;
int *p;
p = &a;
p = &b;
*p = 7;
```

执行这些指令后，变量 a 将保持为 2，变量 b 将变为 7。

在指针声明前加上 const 关键字：

```c
int a = 2, b = 3;
int const *p;
p = &a;
p = &b;
*p = 7;  // 编译错误
```

现在 p 是一个指向常量的指针。尝试编译这段代码将在第 5 行产生编译错误，因为试图修改常量是非法的。

现在将 const 关键字放在稍右的位置：

```c
int a = 2, b = 3;
int *const p;
```

```
p = &a;  // 编译错误
p = &b;  // 编译错误
*p = 7;
```

现在 p 是一个常量指针。在第 3 和 4 行会得到编译错误，因为不能改变指针本身。但是可以修改指针所指向的内存，所以第 5 行不会有编译错误。

6.3 指针与数组的关系

注意，指针的声明和堆上动态数组的声明是相同的。实际上，在动态数组的声明中：

```c
int *a = malloc(sizeof(int) * size);
```

变量 a 是一个指针，包含数组起始位置的地址。因此，解引用 *a 将给我们数组的第零个元素。此外，操作 a+1 被编译器转换为 a+sizeof(类型)，因此，a+1 是指向数组第一个元素的指针，而*(a+1)是数组的第一个元素本身。

代码清单 23：数组与指针的二重性

```c
#include <stdio.h>
#define SIZE 10

int main()
{
    int arr[SIZE];
    for (int i = 0; i < SIZE; i++)
    {
        arr[i] = i;
    }
    int *p = arr;
    printf("%d\n", arr[4]);
    printf("%d\n", p[4]);
    printf("%d\n", *(p + 4));
    printf("%d\n", *(arr + 4));
    printf("%p\n", p + 4);
    printf("%p\n", arr + 4);
    printf("%d\n", ++*p);
    printf("%d\n", ++*arr);
    printf("%d\n", *++p);
    printf("%ld\n", p-arr);
    // printf("%d\n", *++arr);
    return 0;
}
```

输出：

```
4
4
4
4
0x7ffdafc0e280
0x7ffdafc0e280
1
2
1
1
```

说明：

1. 第 12、13 行演示了数组名和指针名都可以使用[]运算符，并且会产生相同的结果。
2. 通过将整数加到数组名或指针上，然后解引用获得的指针，可以达到相同的效果（第 14、15 行）。
3. 第 16 和 17 行显示表达式 p+4 和 arr+4 导致相同的地址。
4. 在第 18 行，我们获取数组第零个元素的指针，解引用它并加 1。因为 p[0]等于零，所以第 18 行将输出数字 1。
5. 在第 19 行，我们再次获取数组第零个元素的指针，并做与第 18 行相同的操作。但是因为在第 18 行我们已经将 p[0]增加了 1，所以现在得到的答案是 2。
6. 在第 20 行，我们首先将 p 增加 1（实际上是 p=p+1），然后解引用获得的指针。因此输出的是 arr[1]==p[0]的值。
7. 在第 21 行，我们计算 p 和 arr 之间的差。因为 p 现在等于 arr+1，所以差值显然是 1。
8. 被注释掉的第 22 行说明，尽管数组和指针相似，但它们并不完全相同：数组名被牢固地绑定到其第零个元素。试图改变它会导致编译错误。

6.4 命令行参数

main 函数从操作系统接收命令行参数，即在终端中运行程序时程序名称后面写入的字符串。main 函数接收 argc（参数数量）和 char* argv[]（指向 char 的指针数组，即字符串数组）。argv[0]通常是可执行文件的名称。数组的后续元素 argv[1]、argv[2]...是命令行参数本身。argv 的大小就是 argc。

代码清单 24 中的程序输出传递给程序的所有命令行参数。

代码清单 24：输出命令行参数

```c
#include <stdio.h>
int main(int argc, char* argv[])
{
    for (int i=1; i<argc; i++)
```

```
    {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

6.5 指向数组的指针和指针数组

前一节的内容激发了我们大胆的实验想法：除了指针数组 char* argv[]，我们还可以用星号和括号构造出 int (*a*)[]这样的结构。*基于指针和数组的二重性，我们可能会认为这两种类型完全等价：char* argv[] = char (*argv)[]，它们都表达了"数组的数组"的概念。

然而，尝试将一个值赋给另一个会遇到编译器的不理解。因此，这些是不同的类型。

代码清单 25 的程序可以帮助我们理清这个问题。

代码清单 25：指向数组的指针和指针数组在内存中的存储方式

C

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char *arr[3];
    arr[0] = calloc(10, sizeof(char));
    arr[1] = calloc(10, sizeof(char));
    arr[2] = calloc(10, sizeof(char));

    for (int i = 0; i < 10; i++)
        arr[1][i] = i;
    for (int i = 0; i < 10; i++)
        arr[1][i] = i * 10;
    for (int i = 0; i < 10; i++)
        arr[1][i] = i * 100;

    char **array_of_pointers = calloc(3, sizeof(char *));
    array_of_pointers[0] = arr[0];
    array_of_pointers[1] = arr[1];
    array_of_pointers[2] = arr[2];

    for (int i = 0; i < 3; i++)
    {
        printf("array_of_pointers[%d] = %p\n",
               i, *(array_of_pointers + i));
        for (int j = 0; j < 3; j++)
            printf("array_of_pointers[%d][%d]=%d\t"
                   "&array_of_pointers[%d][%d]=%p\n",
                   i, j, array_of_pointers[i][j],
                   i, j, &array_of_pointers[i][j]);
    }

    int arr1[2][5] = {{1,2,3,4,5}, {10,20,30,40,50}};
    int (*ptr_to_array)[5];
```

```
    ptr_to_array = &arr1[0];

    for (int i = 0; i < 2; i++)
    {
        printf("Array number = %d\t pointer to array=%p\n",
               i, ptr_to_array);
        for (int j = 0; j < 5; j++)
        {
            printf("Element number = %d\t address = %p\t",
                   j, (*ptr_to_array + j));
            printf("Value = %d\n", *(*ptr_to_array + j));
        }
        ptr_to_array++;
    }

    free(arr[0]);
    free(arr[1]);
    free(arr[2]);
}
```

输出：

```
array_of_pointers[0] = 0x62cebc5c42a0
array_of_pointers[0][0]=0       &array_of_pointers[0][0]=0x62cebc5c42a0
array_of_pointers[0][1]=0       &array_of_pointers[0][1]=0x62cebc5c42a1
array_of_pointers[0][2]=0       &array_of_pointers[0][2]=0x62cebc5c42a2
array_of_pointers[1] = 0x62cebc5c42c0
array_of_pointers[1][0]=0       &array_of_pointers[1][0]=0x62cebc5c42c0
array_of_pointers[1][1]=100     &array_of_pointers[1][1]=0x62cebc5c42c1
array_of_pointers[1][2]=-56     &array_of_pointers[1][2]=0x62cebc5c42c2
array_of_pointers[2] = 0x62cebc5c42e0
array_of_pointers[2][0]=0       &array_of_pointers[2][0]=0x62cebc5c42e0
array_of_pointers[2][1]=0       &array_of_pointers[2][1]=0x62cebc5c42e1
array_of_pointers[2][2]=0       &array_of_pointers[2][2]=0x62cebc5c42e2
Array number = 0         pointer to array=0x7ffc2f9a7080
Element number = 0       address = 0x7ffc2f9a7080       Value = 1
Element number = 1       address = 0x7ffc2f9a7084       Value = 2
Element number = 2       address = 0x7ffc2f9a7088       Value = 3
Element number = 3       address = 0x7ffc2f9a708c       Value = 4
Element number = 4       address = 0x7ffc2f9a7090       Value = 5
Array number = 1         pointer to array=0x7ffc2f9a7094
Element number = 0       address = 0x7ffc2f9a7094       Value = 10
Element number = 1       address = 0x7ffc2f9a7098       Value = 20
Element number = 2       address = 0x7ffc2f9a709c       Value = 30
Element number = 3       address = 0x7ffc2f9a70a0       Value = 40
Element number = 4       address = 0x7ffc2f9a70a4       Value = 50
```

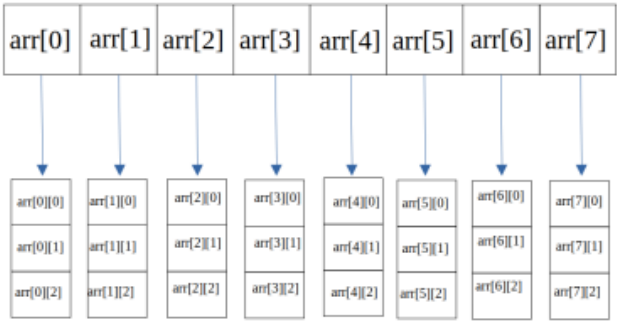


Рис. 5: Массив указателей

Рис. 6: Указатель на массив

通过仔细分析代码和程序的运行结果，我们可以得出以下结论：

1. arr 数组的每个元素都是一个动态数组，其中所有元素连续地一个接一个地排列。这些数组中元素的地址相差一个字节 — char 的大小。同时，一个数组的最后一个元素与下一个数组的第零个元素之间的距离不等于一个字节。
2. 指向指针的指针和指针数组本质上是相同的。
3. 指向数组的指针 ptr_to_array 的所有元素都紧挨着排列，它们的地址相差基本变量的大小，在这种情况下是 int。
4. 指向数组的指针和静态二维数组（静态数组的静态数组）是相同的。

指向数组的指针和指针数组在内存中的布局差异见图 5 和图 6。

因此，指向数组的指针是静态二维数组。指针数组是动态二维数组。

第七章

7 函数

7.1 函数的创建和使用

函数是一个可以接收和返回值的命名代码片段。

函数声明：

```c
return_type function_name(t1 val1, t2 val2, ...tn valn)
```

其中：

- return_type —— 返回值类型
- function_name —— 函数名
- t1, t2, ...tn —— 第 1 个、第 2 个...第 n 个参数的类型
- val1, val2, ..., valn —— 第 1 个、第 2 个...第 n 个参数的名称

函数名称、类型、参数数量和顺序构成函数签名。

函数的定义（即函数"体"，具体执行的指令）可以与声明合并，也可以分开。

如果函数定义和函数声明是分开的，那么在函数声明中可以省略参数名称。

必须在首次使用函数之前声明该函数。但函数的定义可以在代码中首次使用之后。

使用关键字 return 从函数返回值。执行 return 指令后，函数终止，不再执行其他操作。

示例 26：函数声明和定义合并

```c
#include <stdio.h>
int add(int a, int b)
{
    return a + b;
}
int main()
{
    printf("Sum of 2 and 3 is %d\n", add(2, 3));
    return 0;
}
```

示例 27：函数声明和定义分开

```c
```

```c
#include <stdio.h>
int add(int, int);
int main()
{
    printf("Sum of 2 and 3 is %d\n", add(2, 3));
    return 0;
}
int add(int a, int b)
{
    return a + b;
}
```

7.2 通过指针传递参数

让我们编写一个 swap 函数，用于交换两个整数参数的值。第一次尝试创建这样的函数如示例 28 所示。很容易发现这种尝试并不成功：调用函数后变量的值并没有交换。原因在于 swap 函数处理的是传递给它的变量的副本。函数的参数和局部变量存储在栈上，函数结束工作后会被删除。因此，函数对传递给它的参数所做的所有更改都是在副本上进行的，这些副本在函数工作结束后立即被删除，而参数本身保持不变。

示例 28：交换变量的失败尝试

C

```c
#include <stdio.h>
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
int main()
{
    int a = 2, b = 3;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b); // a = 2, b = 3
    return 0;
}
```

解决方案：向函数传递变量的指针。这样，传递的指针将被复制到函数中，通过解引用这些指针，我们就能修改原始变量！因此，在调用函数时，我们将传递变量的地址。这个想法在示例 29 中实现。

示例 29：成功交换变量

C

```c
#include <stdio.h>
int swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```c
int main()
{
    int a = 2, b = 3;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b); // a = 3, b = 2
    return 0;
}
```

现在，特别是，我们明白为什么在 scanf 函数中需要在变量名前写&符号：实际上，指向变量的指针被传递给这个函数，这样 scanf 函数就能够将更改写入传递的参数中。

7.3 函数指针

函数指针包含函数命令所在的地址。类似于数组，函数名是指向函数开始处的指针。

示例 30：将函数作为参数传递

C

```c
#include <stdio.h>
double f(double x)
{
    return x * x;
}
double g(double x)
{
    return x;
}
double Integral(double a, double b, int n,
                double (*f)(double x))
{
    double h = (b - a) / n;
    double sum = 0.5 * (f(a) + f(b));
    for(int i=1; i<n; i++)
    {
        double x = a+i*h;
        sum += f(x);
    }
    sum *= h;
    return sum;
}
int main()
{
    printf("%lf\n", Integral(0, 3, 1000, f));
    printf("%lf\n", Integral(0, 3, 1000, g));
    return 0;
}
```

7.4 可变参数函数

7.5 调用栈

假设我们有两个函数 foo()和 bar()。假设从 main 调用 foo()，从 foo()调用 bar。那么栈可以用图 7 示意表示。

让我们看示例 31 中的程序。当调用函数 f 时，栈将呈现如图 8 所示的形式。

Frame pointer 指向当前执行函数的栈的开始。

Stack pointer 指向当前函数栈的末尾。

从上面数第 3 和第 4 个单元格中的 2, 1 是接收的值（副本）。

retval —— 返回值。

retaddress —— 返回地址（函数完成工作后要去的地方）。

registers —— 调用函数的处理器寄存器内容。

示例 31 栈的演示

```c
#include <stdio.h>
int f(int x,int y){
  int d = x+y;
  int s = x-y;
  return d + s ;
}
int main (){
  int a = 1;
  int b = 2;
  int z = f(a,b);
  return 0 ;
}
```

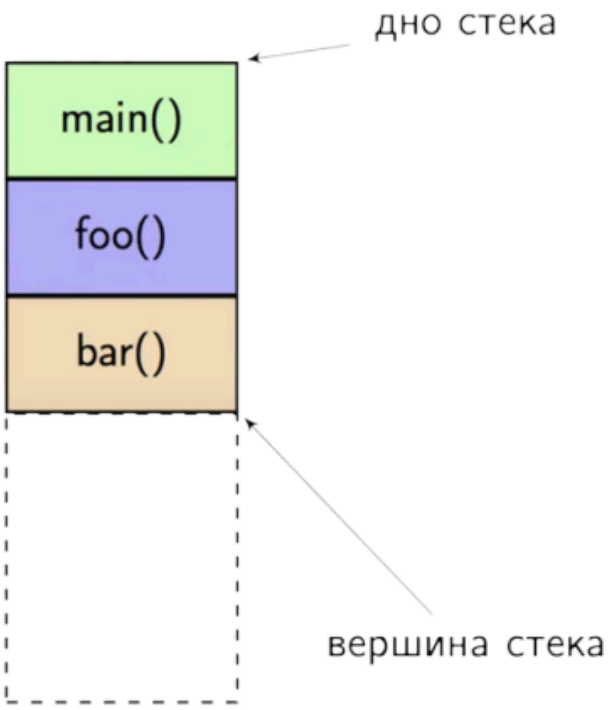


Рис. 7: Стек вызовов

示例 32 缓冲区溢出

```c
#include <stdio.h>
int MaliciousCode(int x, int y){
  printf ("Hey ! \ n") ;
  return 0;
}
int GoodCode(){
  int∅  m[1];
  m[3] = (int∅)MaliciousCode;
  return 0;
}
int main (){
  GoodCode ();
  return 0;
}
```

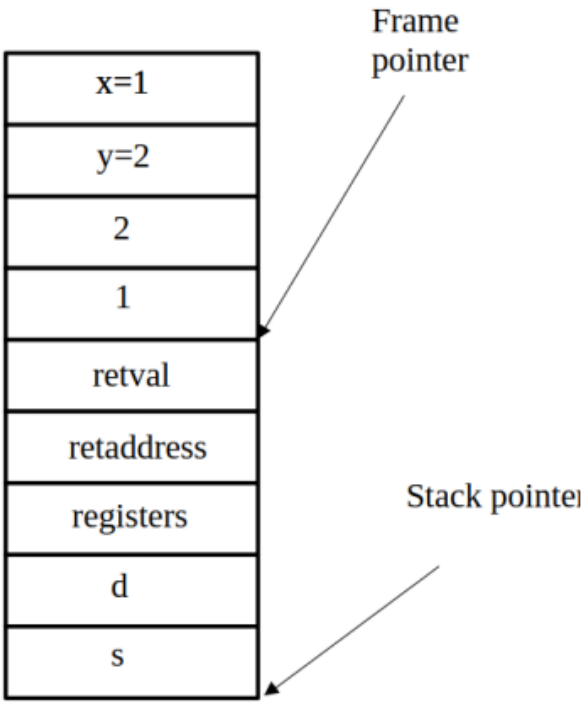


Рис. 8: Структура стека

7.6 在函数中传递数组和从函数返回数组

任务 1：创建和打印数组

编写一个名为 `CreateArray` 的函数，创建长度为 n 的数组并填入随机数。编写一个名为 `PrintArray` 的函数，将数组元素显示在屏幕上。

解决方案如下：

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

int* CreateArray(int n)
{
    int* a = calloc(n, sizeof(int));
    for(int i = 0; i < n; i++)
        a[i] = rand() % 10;
    return a;
}

void PrintArray(const int* a, int n)
{
    for(int i=0; i<n; i++)
```

```
        printf("%d ", a[i]);
    printf("\n");
}

int main()
{
    int n;
    scanf("%d", &n);
    int* arr = CreateArray(n);
    PrintArray(arr, n);
    free(arr);
    return 0;
```

注意：在第 12 行中使用了指向常量的指针，这样如果代码中意外出现类似 `a[i] = 777;` 的指令时，编译器会报错。

任务 2：矩阵向量乘法

方案 1：使用静态二维数组

C

```c
#include <stdio.h>
#include <stdlib.h>
#define ROWS 5
#define COLS 4

int CreateVector(int a[])
{
    for(int i = 0; i < COLS; i++)
        a[i] = rand() % 10;
}

void PrintVector(const int* a, int n)
{
    printf("Vector:\n");
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void CreateMatrix(int (*mat)[COLS])
{
    for(int i = 0; i < ROWS; i++)
        for(int j = 0; j < COLS; j++)
            mat[i][j] = rand() % 10;
}

void PrintMatrix(int (*mat)[COLS])
{
    printf("Matrix:\n");
    for(int i = 0; i < ROWS; i++)
```

```c
    {
        for(int j = 0; j < COLS; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

void MatVecMult(int (*mat)[COLS], int* vec, int* res)
{
    for(int i = 0; i < ROWS; i++)
    {
        res[i] = 0;
        for(int j = 0; j < COLS; j++)
            res[i] += mat[i][j] * vec[j];
    }
}

int main()
{
    int vec[COLS];
    int mat[ROWS][COLS];
    int res[ROWS];
    CreateVector(vec);
    PrintVector(vec, COLS);
    CreateMatrix(mat);
    PrintMatrix(mat);
    MatVecMult(mat, vec, res);
    PrintVector(res, ROWS);
    return 0;
}
```

重要说明

1. 根据第 51 页的结论，作为二维静态数组传递给函数时，可以使用 `int mat[][COLS]` 或 `int (*mat)[COLS]` 两种形式。
2. 应避免编写如下代码：

```c
int* CreateVector()
{
    int a[COLS];
    for(int i = 0; i < COLS; i++)
        a[i] = rand() % 10;
    return a;
}
```

这种情况下返回了局部变量 `a` 的地址，该变量位于 `CreateVector` 函数的栈上，函数结束时会被销毁。

方案 2：使用动态内存分配

```c
```

```c
#include <stdio.h>
#include <stdlib.h>
#define ROWS 5
#define COLS 4

int* CreateVector(int n)
{
    int* a = calloc(n, sizeof(int));
    for(int i = 0; i < COLS; i++)
        a[i] = rand() % 10;
    return a;
}

int** CreateMatrix(int r, int c)
{
    int** mat = calloc(r, sizeof(int*));
    for(int i = 0; i < r; i++)
    {
        mat[i] = calloc(c, sizeof(int));
        for(int j = 0; j < c; j++)
            mat[i][j] = rand() % 10;
    }
    return mat;
}

int* MatVecMult(int** mat, int* vec, int r, int c)
{
    int* res = calloc(r, sizeof(int));
    for(int i = 0; i < r; i++)
    {
        res[i] = 0;
        for(int j = 0; j < c; j++)
            res[i] += mat[i][j] * vec[j];
    }
    return res;
}

int** DeleteMatrix(int*** mat, int r)
{
    for(int i=0; i<r; i++)
    {
        free((*mat)[i]);
    }
    free(*mat);
    return NULL;
}

int* DeleteVector(int** vec)
{
    free(*vec);
    return NULL;
}
```

```
// 主函数实现...
```

这个实现展示了：

1. 正确的动态内存分配方法
2. 适当的内存释放机制
3. 安全的矩阵和向量操作
4. 返回值的正确处理

7.7 复杂声明

第八章

8 字符串

8.1 常量指针和指向常量的指针

让我们看看以下代码片段：

```c
int a = 2, b = 3;
int* p;
p = &a;
p = &b;
*p = 7;
```

执行这些指令后，变量 a 中的值为 2，变量 b 中的值为 7。

如果在指针声明前添加 const 关键字：

```c
int a = 2, b = 3;
int const* p;
p = &a;
p = &b;
*p = 7;  // 编译错误
```

现在 p 是一个指向常量的指针。尝试编译这段代码会在第 5 行产生编译错误，因为试图修改常量是不合法的。

如果我们将 const 关键字放在稍右的位置：

```c
int a = 2, b = 3;
int* const p;
p = &a;   // 编译错误
p = &b;   // 编译错误
*p = 7;   // 允许
```

现在 p 是一个常量指针。在第 3 和第 4 行会得到编译错误，因为不能改变指针本身。但是可以修改它所指向的内存，所以第 5 行不会出现编译错误。

8.2 什么是字符串

字符串是一个字符数组（char 类型的元素），其中最后一个字符必须是代码零（数字 0，或'\0'）。

示例 35：字符串声明

```c
```

```c
const char* s1 = "Hello, world!";
char s2[100] = "Hello, world!";
char* s3 = malloc(100);
s1 = s2;    // 正确
s2 = s1;    // 编译错误
s3 = s1;    // 内存泄漏
s1[0] = 'h'; // 编译错误
s2[0] = 'h'; // 正确
printf("%d\n", sizeof(s1));  // 8
printf("%d\n", sizeof(s2));  // 100
free(s3);
```

示例 35 的说明：

1. 第 1 行声明了一个指向 char 的指针，用字符串字面量初始化。这个字符串字面量被放在初始化数据段的只读部分（见第 34 页）。这就是为什么第 7 行的赋值会失败：如果在第 1 行 s1 被声明为指向常量的指针，这将是编译错误；如果省略 const 修饰符，则是运行时错误。
2. 第 4 行我们利用指针和数组的二重性，这允许我们将数组名赋给指针。
3. 第 5 行我们试图进行相反的赋值，这会导致编译错误。原因是数组名与它所指向的内存区域是"牢固绑定"的。
4. 第 6 行的赋值导致内存泄漏，因为我们失去了对第 3 行通过 malloc 分配的内存区域的访问。此外，第 11 行的 free 将试图释放只读数据段，这是一个糟糕的想法，也会导致运行时错误。
5. 第 9 和第 10 行说明了数组和指针的另一个区别：在 64 位系统中指针的大小是 8 字节，而数组的大小等于整个数组的字节数。

表 2:基本字符串函数表

函数名	功能
strcpy(dest, source)	将 source 复制到 tt dest
strcat(dest, source)	将 source 追加到 dest 末尾
strlen(s)	返回 s 的长度（不包括'\0'）
strcmp(s1, s2)	比较 s1 和 s2，返回 0、1、–1
strchr(str, ch)	在字符串中查找字符
strstr(str, substr)	在字符串中查找子串
strspn(s1, s2)	查找 s1 中由 s2 中字符组成的最大前缀长度
strcspn(s1, s2)	查找 s1 中不含 s2 中字符的最大前缀长度
strpbrk(s1, s2)	查找 s1 中第一次出现的任何 s2 中的字符

8.3 字符串操作函数

基本的字符串处理函数如表 2 所示。

家庭作业：在网站上查看表 2 中所列函数的语法和示例。

示例 36 字符串函数的实现示例

C

```c
// 获取字符串长度
int str_len(char const* str)
{
    int len = 0;
    while(str[len++]);
    return len-1;
}

// 字符串复制
void str_copy(char* dest, char const* source)
{
    while(*dest++ = *source++);
}

// 字符串连接
void str_concat(char* dest, char const* source)
{
    while(*dest++);
    str_copy(dest-1, source);
}

// 字符串比较
int str_compare(char const* str1, char const* str2)
{
    while(*str1 && *str2)
    {
        if(*str1 > *str2)
            return 1;
        if(*str1 < *str2)
            return -1;
        str1++;
        str2++;
    }
    if(*str1)
        return 1;
    if(*str2)
        return -1;
    return 0;
}
```