C++ 面向对象

极夜酱

目录

Chapter 1 C++ 基础

1.1 数据类型

1.1.1 C++ 简介

C++ 由 Bjarne Stroustrup 于 1979 年在贝尔实验室发明，C++ 在 C 语言的基础上引起并扩充了面向对象的概念，最初命名为带类的 C（C with classes），后更名为 C++。C++ 应用非常广泛，常用于系统开发、引擎开发等领域，支持类、封装、继承、多态等特性。

Hello World!

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

运行结果

```
Hello World!
```

头文件 iostream 库声明了用于在标准输入输出设备上操作的对象，头文件中包含 cin（标准输入流）、cout（标准输出流）、cerr（标准错误流）、clog（标准日志流）对象。

using namespace std 告诉编译器使用 std 命名空间，命名空间是 C++ 中新的概念。

1.1.2 关键字

C++ 中的关键字不能作为常量名、变量名、或其它标识符名称。

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

表 1.1: 关键字

1.1.3 数据类型

一些基本数据类型可以使用 signed 和 unsigned 修饰符进行修饰。

类型	占用内存	范围
char	1 字节	$-128 \sim 127$
unsigned char	1 字节	$0 \sim 255$
signed char	1 字节	$-128 \sim 127$
int	4 字节	$-2147483648 \sim 2147483647$
unsigned int	4 字节	$0 \sim 4294967295$
signed int	4 字节	$-2147483648 \sim 2147483647$
short	2 字节	$-32768 \sim 32767$
unsigned short	2 字节	$0 \sim 65535$
signed short	2 字节	$-32768 \sim 32767$
long	4 字节	$-2147483648 \sim 2147483647$
signed long	4 字节	$0 \sim 4294967295$
unsigned long	4 字节	$-2147483648 \sim 2147483647$
long long	8 字节	$-9223372036854775808 \sim 9223372036854775807$
signed long long	8 字节	$-9223372036854775808 \sim 9223372036854775807$
unsigned long long	8 字节	$0 \sim 18446744073709551615$
float	4 字节	$-3.4e38 \sim 3.4e38$
double	8 字节	$-1..7e308 \sim 1.7e308$

表 1.2: 数据类型

数据类型

```cpp
#include <iostream>
#include <limits>
using namespace std;

int main() {
    cout << "int" << endl;
    cout << "size: " << sizeof(int) << endl;
    cout << "min: " << numeric_limits<int>::min() << endl;
    cout << "max: " << numeric_limits<int>::max() << endl;
    cout << "--------------------------------" << endl;
    cout << "short" << endl;
```

```
12      cout << "size: " << sizeof(short) << endl;
13      cout << "min: " << numeric_limits<short>::min() << endl;
14      cout << "max: " << numeric_limits<short>::max() << endl;
15      cout << "----------------------------------" << endl;
16      cout << "long" << endl;
17      cout << "size: " << sizeof(long) << endl;
18      cout << "min: " << numeric_limits<long>::min() << endl;
19      cout << "max: " << numeric_limits<long>::max() << endl;
20      cout << "----------------------------------" << endl;
21      cout << "long long" << endl;
22      cout << "size: " << sizeof(long long) << endl;
23      cout << "min: "
24          << numeric_limits<long long>::min() << endl;
25      cout << "max: "
26          << numeric_limits<long long>::max() << endl;
27      cout << "----------------------------------" << endl;
28      cout << "float" << endl;
29      cout << "size: " << sizeof(float) << endl;
30      cout << "min: " << numeric_limits<float>::min() << endl;
31      cout << "max: " << numeric_limits<float>::max() << endl;
32      cout << "----------------------------------" << endl;
33      cout << "double" << endl;
34      cout << "size: " << sizeof(double) << endl;
35      cout << "min: " << numeric_limits<double>::min() << endl;
36      cout << "max: " << numeric_limits<double>::max() << endl;
37      cout << "----------------------------------" << endl;
38      cout << "char" << endl;
39      cout << "size: " << sizeof(char) << endl;
40      cout << "----------------------------------" << endl;
41      cout << "bool" << endl;
42      cout << "size: " << sizeof(bool) << endl;
43      cout << "min: " << numeric_limits<bool>::min() << endl;
44      cout << "max: " << numeric_limits<bool>::max() << endl;
45      cout << "----------------------------------" << endl;
46
47      return 0;
48  }
```

1.2 命名空间

1.2.1 命名空间（namespace）

多人共同开发时，经常会出现变量和函数定义冲突，例如编写了相同名称的函数。命名空间用于解决这个问题，它可以作为附加信息来区分不同库中的变量和函数。

namespace 表示了标识符的可见范围，C++ 标准库中的所有标识符都定义于一个名为 std 的 namespace 中。如果不使用 using namespace std，就必须使用 std::cout、std::endl 等表示其中的标识符。

命名空间

```cpp
#include <iostream>

namespace foo {
    int n = 123;
    void f() {
        std::cout << "foo::f()" << std::endl;
    }
};

namespace bar {
    int n = 456;
    void f() {
        std::cout << "bar::f()" << std::endl;
    }
};

int main() {
    std::cout << foo::n << std::endl;
    foo::f();
    std::cout << bar::n << std::endl;
    bar::f();
```

```
22      return 0;
23  }
```

运行结果

```
123
foo::f()
456
bar::f()
```

1.3 结构体与共同体

1.3.1 结构体

结构体是一种用户自定义的数据类型，它允许存储不同类型的数据项。结构体的声明使用关键字 struct，声明通常定义为全局变量，这样就可以被多个函数所使用的了。

通过成员运算符【.】可以访问一个结构体之中的成员变量。

关键字 typedef 可以用来给数据类型定义别名，通过使用 typedef 可以简化结构体的声明，不用每次都加上 struct 关键字了。

结构体

```
#include <iostream>

using namespace std;

typedef struct {
    int year;
    int month;
    int day;
} Date;

int main() {
    Date date;
    date.year = 2021;
    date.month = 8;
    date.day = 11;
    cout << date.year << "/"
        << date.month << "/"
        << date.day << endl;
    return 0;
}
```

运行结果

```
2021/8/11
```

结构体也可以作为函数参数进行传递。如果是按值传递，那么在函数中会新创建一个结构体变量，并复制调用者的结构体的值。如果是按址传递，则需要传递结构体的指针。

间接引用运算符【->】可以直接访问结构体指针所指的结构变量中的成员。

1.3.2 共同体

在存储多个成员信息时，编译器会自动给 struct 每个成员分配存储空间，struct 可以存储多个成员信息。而 union 每个成员会用同一个存储空间，只能存储一个成员的信息。

在任何同一时刻，union 只存放了一个被先选中的成员，而结构体的所有成员都存在。对于 ‘union‘的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于 struct 的不同成员赋值是互不影响的。

1.4 常量指针与指针常量

1.4.1 常（量）指针

常（量）指针是指在定义指针变量时，在数据类型前用 const 修饰。

定义一个常指针后，就不能通过指针去更改所指向的变量的值，但是指针的指向可以改变。

常指针

```
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    const int *p = &a;  // 等价于int const *p
    *p = 456;
    return 0;
}
```

运行结果

```
error: assignment of read-only location '* p'
```

1.4.2 指针常量

指针常量是指在定义指针时，在指针变量前用 const 修饰。

指针常量不允许修改，必须要在定义时初始化，之后不能修改指针的指向。

指针常量

```cpp
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    int b = 456;
    int * const p = &a;
    p = &b;
    return 0;
}
```

运行结果

```
error: assignment of read-only variable 'p'
```

1.4.3 常指针常量

常指针常量表示指针的指向不能改变，同时指针指向的值也不能改变。

常指针常量

```cpp
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    int b = 456;
    const int * const p = &a;   // 等价于int const * const p
    *p = 789;
    p = &b;
    return 0;
}
```

运行结果

```
error: assignment of read-only location '*(const int*)p'
error: assignment of read-only variable 'p'
```

1.5 内联函数

1.5.1 内联函数（Inline Function）

内联函数是在编译时期展开，不用执行进入函数的步骤，直接执行函数体，相当于把内联函数里面的内容写在调用内联函数处。编译器一般不内联包含循环、递归、switch 等复杂操作的函数。是否内联，程序员不可控，内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

内联函数

```cpp
#include <iostream>

using namespace std;

inline int max(int num1, int num2) {
    return num1 > num2 ? num1 : num2;
}

int main() {
    cout << max(92, 28) << endl;
    return 0;
}
```

运行结果

```
92
```

编译器对内联函数的处理首先是将内联函数复制到调用处，并为局部变量分配内存，将输入参数和返回值映射到局部变量空间中。如果内联函数有多个返回点，将会使用 goto 语句跳转到代码块的末尾。

使用内联函数的优点在于同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。并且内联函数相

比宏函数来说，在代码展开时，会做安全检查或自动类型转换，而宏定义则不会。在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量。

但是内联是以代码膨胀（复制）为代价，消除函数调用带来的开销，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。并且内联函数无法随着函数库的升级而升级，内联函数的改变需要重新编译，不像普通函数可以直接链接。

1.6 重载函数

1.6.1 函数默认参数

在进行函数参数定义的时候，也可以设置默认值。当参数没有传递的时候就利用默认值来进行参数内容的填充，如果在参数上定义了默认值，那么该参数一定要放在参数列表的最后。

函数默认参数

```
#include <iostream>

using namespace std;

void setDate(int year = 1970, int month = 1, int day = 1) {
    cout << year << "/" << month << "/" << day << endl;
}

int main() {
    setDate(2021, 8, 15);
    setDate(2021, 7);
    setDate(2021);
    setDate();
    return 0;
}
```

运行结果

```
2021/8/15
2021/7/1
2021/1/1
1970/1/1
```

1.6.2 重载函数

重载（overload）表示在同一个作用域中声明了一个与之前声明过的函数具有相同名称的函数，但是它们的参数列表不同。当调用一个重载函数时，编译器通过传递的参数类型，选用最合适的定义。

重载函数

```
#include <iostream>
using namespace std;

int max(int num1, int num2) {
    return num1 > num2 ? num1 : num2;
}

double max(double num1, double num2) {
    return num1 > num2 ? num1 : num2;
}

char max(char num1, char num2) {
    return num1 > num2 ? num1 : num2;
}

int main() {
    cout << max(2, 8) << endl;
    cout << max(3.14, 2.71) << endl;
    cout << max('H', 'D') << endl;
    return 0;
}
```

运行结果

```
8
3.14
H
```

1.7 引用

1.7.1 引用（Reference）

在 C 语言中【&】表示取地址符，但是在 C++ 中还有引用的特性。引用是指对某一变量的别名，因此引用本身并不占存储单元，对引用的操作与直接对变量的操作完全一样。

```
data_type &ref_var = var;
```

声明引用时必须对其初始化，并且之后不能再将该引用作为其它变量的别名。

引用的一个重要作用就是作为函数的参数。C 语言中函数参数传递是按值传递，如果有大块数据作为参数传递的时候，往往采用指针的方式，因为这样可以避免将整块数据全部压栈，可以提高程序的效率。但是 C++ 中传递引用给函数与传递指针的效果是一样的，使用引用传递函数的参数，在内存中并没有产生实参的副本，而是直接对实参操作。

引用

```
#include <iostream>

using namespace std;

void swap(int& num1, int& num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}

int main() {
    int a = 92;
    int b = 28;
    cout << "Before: " << a << " " << b << endl;
    swap(a, b);
```

```
16      cout << "After: " << a << " " << b << endl;
17      return 0;
18  }
```

运行结果

```
Before: 92 28
After: 28 92
```

指针与引用的区别：

- 指针有自己的一块空间，而引用只是一个别名，对引用的操作等效于对原变量的操作。
- 指针的大小为 4 或 8（根据 OS），而引用则是被引用对象的大小。
- 指针可以被初始化为 NULL，而引用必须被初始化且必须是一个已有对象的引用。
- 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象。
- 引用在初始化后，便不能再更改引用的目标。
- 指针可以有多级指针，而引用只有一级。
- 指针和引用使用【++】运算符的意义不一样。
- 返回动态内存分配的内存时必须使用指针，引用可能引起内存泄漏。

1.8 函数指针

1.8.1 函数指针

函数指针即指向函数的指针，它最大的作用是把一个函数作为参数传递给另外一个函数。在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。

函数指针

```
#include <iostream>

using namespace std;

int add(int num1, int num2) {
    return num1 + num2;
}

int sub(int num1, int num2) {
    return num1 - num2;
}

int get(int num1, int num2, int (*calculate)(int num1, int num2)) {
    return calculate(num1, num2);
}

int main() {
    cout << get(7, 3, add) << endl;
    cout << get(7, 3, sub) << endl;
    return 0;
}
```

运行结果

```
10
4
```

1.8.2 回调函数（Callback Function）

假设公司要开发一款主打解决国民早餐问题的 APP 明日油条，为了加快开发进度，这款应用由 A 小组和 B 小组协同开发。

其中有一个核心模块由 A 小组开发供 B 小组调用，这个核心模块被写成了一个函数 make_youtiao()。

如果 make_youtiao() 执行地很快并可以立即返回，那么 B 小组只需：

1. 调用 make_youtiao()
2. 等待函数执行完成
3. 执行后续流程 sell() 出售

但是现实中 make_youtiao() 函数需要处理的数据非常庞大，例如 make_youtiao(10000) 不会立刻返回，而是需要等待 10 分钟才能执行完成。显然直接调用的话，需要等待 10 分钟后才能继续执行后续代码，这并不是一种高效的做法。

因此一种更好的做法是调用 make_youtiao() 后不再等待这个函数执行完成，而是让这个函数知道制作完油条后该干什么，例如“制作 10000 个油条，然后卖出去”。

因此 make_youtiao() 需要增加一个参数，除了指定制作油条的数量外，还可以指定制作好后该干什么，这个被 make_youtiao() 调用的函数就叫回调。

回调函数

```
1 #include <iostream>
2
```

```cpp
using namespace std;

void sell(int n) {
    cout << "selling youtiao ..." << endl;
}

void make_youtiao(int n, void (*sell)(int)) {
    cout << "making youtiao ..." << endl;
    for(int i = 0; i < n; i++);
    sell(n);
}

int main() {
    make_youtiao(10000, sell);
    return 0;
}
```

运行结果

```
making youtiao ...
selling youtiao ...
```

1.9 内存管理

1.9.1 malloc() / free()

C 语言使用 malloc() 分配内存，使用 free() 释放已分配的内存。当 malloc() 分配的内存过多/不够时，可以使用 realloc() 重新分配内存。

malloc() 的功能是向系统申请指定的内存空间（以字节为单位），使用该函数需要包含头文件 stdlib.h。

```
1 void* malloc(size_t size);
```

malloc() 的返回值为 void * 类型，表示一个指向申请到的空间的首地址，是一个无类型的指针，开发者需要自行转换为自己需要的类型。如果 malloc() 申请内存失败，则会返回空指针 NULL。

通过 malloc() 申请来的空间是需要归还给操作系统的，否则程序长时间运行内存会逐渐下降。通过 free() 可以把申请来的空间释放，但是有两点需要注意：

1. 只能释放通过 malloc() 申请得到的空间。
2. 只能通过空间的首地址进行释放。

malloc() / free()

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main() {
7      const int n = 10;
8      int *arr = (int *)malloc(sizeof(int) * n);
9      if(!arr) {
10         cerr << "Memory allocation failed." << endl;
11         return 1;
```

```
12      }
13      for(int i = 0; i < n; i++) {
14          arr[i] = i;
15          cout << arr[i] << " ";
16      }
17      cout << endl;
18      free(arr);
19      return 0;
20  }
```

运行结果

```
0 1 2 3 4 5 6 7 8 9
```

1.9.2 new / delete

C++ 中可以通过 new 运算符实现动态内存分配，如果空间内配失败，程序则抛出 bad_alloc 异常。

```
1  T *p = new T;
2  T *p = new T[N];
```

其中 T 为任意类型名，p 是类型为 T * 的指针。

通过 new 动态内存分配所得的空间在使用完后需要使用 delete 运算符释放空间。

```
1  delete p;
2  delete[] p;
```

如果动态分配 new 一个数组，但是却用 delete 释放，而不是 delete[]，会导致动态分配的数据没有被完全释放。如果动态申请的内存空间不再使用后没有及时释放，就会导致内存泄漏（memory leak）。

new / delete

```cpp
#include <iostream>

using namespace std;

int main() {
    int *p1 = new int;
    *p1 = 928;
    cout << "*p1 = " << *p1 << endl;

    const int n = 10;
    int *p2 = new int[n];
    cout << "p2: ";
    for(int i = 0; i < n; i++) {
        p2[i] = i;
        cout << p2[i] << " ";
    }
    cout << endl;

    delete p1;
    delete[] p2;
    return 0;
}
```

运行结果

```
*p1 = 928
p2: 0 1 2 3 4 5 6 7 8 9
```

1.9.3 malloc() / free() 与 new / delete 区别

- new 分配内存按照数据类型，而 malloc() 按照指定指定大小。
- new 返回指定对象的指针，而 malloc() 返回 void *，因此 malloc() 的返回值一般都需要类型转换。
- new 分配的空间要用 delete 销毁，而 malloc() 要用 free() 销毁。

- new 是一个操作符，malloc() 是一个库函数。
- new 分配失败抛出 bad_alloc 异常，而 malloc() 失败返回 NULL。

1.9.4 内存管理

虚拟内存中栈区的内存是由系统自动分配的，不需要程序员对其进行管理，一般用于存储函数的返回地址、参数、局部变量和返回值。而堆区的内存是动态分配的，并且需要手动释放内存。

栈是运行时的单位，堆是存储的单位。栈解决程序运行问题（如何执行），堆解决数据存储的问题（如何存储）。

内存通常包括了栈区（stack）、堆区（heap）、数据区、程序代码区：

- 栈区：由编译器自动分配和释放，存放函数的参数值、局部变量的值等。
- 堆区：一般由程序员分配和释放，若程序员不释放，程序结束后被 OS 回收。
- 数据区：存放全局变量和静态变量，程序结束后由系统释放。
- 程序代码区：存放函数体的二进制代码。

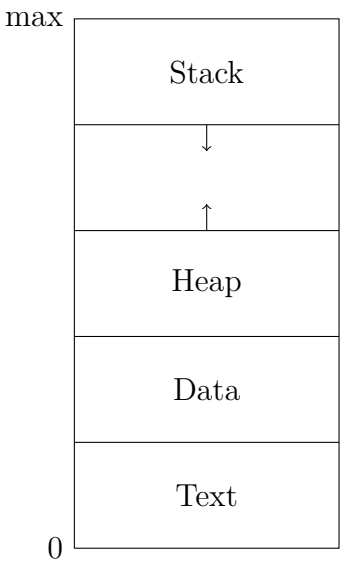

图 1.1: 内存管理