Lecture 2

TOC

- 1. C++ 介绍
- 2. Hello, World!
- 3. 声明和定义
- 4. Statement
- 5. 基础数据类型
- 6. 常量和字面量
- 7. 指针和引用
- 8. 复合数据类型
- 9. Control Flow
- 10. Expression

C++ 介绍

历史

C++由Bjarne Stroustrup于1979年在C语言的基础上开发,目的是通过引入面向对象编程(OOP)的概念,如类、继承和多态,来增强C语言在复杂软件开发中的适应性。

特点

- 灵活:支持面向过程、面向对象、泛型等多种编程范式
- 高效:C++是一种静态类型、编译式编程语言。
- 和 C 语言兼容: C++ 代码可以无缝的和 C 代码进行链接
- 向下兼容:新版本的 C++ 语言标准会向下兼容旧版本的代码

标准

- C++98: 第一个标准化版本
- C++03:对 C++98 的小修小补
- C++11:引入了许多新特性,是的 C++ 更加现代化
- C++14, 17, 20, 23:进一步完善和扩展了 C++ 语言的功能

Hello, World!

一个简单的 C++ 程序

```
#include <iostream>

int main() {

std::cout << "Hellor, World!\n";

return 0;
}</pre>
```

编译运行:

```
g++ hello_world.cpp -o hello_world
./hello_world
```

关键字&标识符

关键字是 C++ 中具有特殊含义的保留字,它们用于表示语言的语法结构和控制指令,例如 int, if, for, return 等。由于关键字已经被语言定义,不能用作变量名、函数名或其他标识符。

标识符是程序员为变量、函数、类、对象等实体自定义的名称。标识符必须遵循一定的规则,如以字母或下划 线开头,不能与关键字冲突,并且在命名时最好具有语义性,以便提高代码的可读性和可维护性。

声明和定义

- **声明**是指告诉编译器某个变量、函数或类的存在。
- **定义**是指指定函数或变量的具体实现,或者为变量分配内存空间。

单定义原则(ODR): C++ 中,单个翻译单元内的非 inline 函数/变量只能出现一次定义

Reference

Statement

语句是依序执行的 C++ 程序片段,主要有以下几种:

- 1. 声明语句
- 2. 带标号语句
- 3. 表达式语句
- 4. 复合语句
- 5. 选择语句
- 6. 循环语句
- 7. 跳转语句
- 8. try 块

Reference:

https://en.cppreference.com/w/cpp/language/statements

基础数据类型

C++ 中的基本数据类型包括整型、浮点型、字符型、布尔型等。

运行时,数据以二进制形式存储在内存中,不同的数据类型占用不同的内存空间,有不同的取值范围和精度。

不同的 CPU 架构存储数据的方式不同,分为大端序和小端序。大端序是指高位字节存储在低地址,小端序是指高位字节存储在高地址。

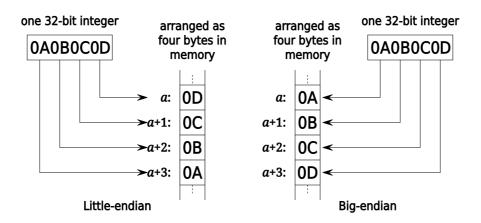


Image Citation: https://en.wikipedia.org/wiki/Endianness

整型
类型

char

short

int

long

long long

类

bool

关键字

bool

char

short

int

long

long long

字节数

1

1

2

4

4

8

取值范围

true/false

 $-2^7 \sim 2^7 - 1$

 $-2^{15} \sim 2^{15} - 1$

 $-2^{31} \sim 2^{31} - 1$

 $-2^{31} \sim 2^{31} - 1$

 $-2^{63} \sim 2^{63} - 1$

无符号整型为上述整型类型前加 unsigned 关键字,取值范围为 $0\sim 2^n-1$,其中 n 为整型字节数。有符号整形以 signed 关键字修饰,可省略。

注:不同CPU架构/编译器下,数据类型的字节数可能有所不同

定长整型(Since C++11)

C++11 引入了定长整型,如 int8_t, uint8_t, int16_t, uint16_t 等,确保了数据类型的字节数,保证了跨平台的兼容性 Reference

存储格式

有符号整型采用补码存储,无符号整型采用原码存储。

算术溢出

整型溢出是指整型变量的值超出了其数据类型所能表示的范围,导致结果不准确。例如, int 类型的变量最大值为 $2^{31}-1$,当其值超过这个范围时,会发生溢出,变为负数。

浮点型

类型

float	float	4	3.4e-38 ~ 3.4e38
double	double	8	1.7e-308 ~ 1.7e308
long double	long double	12	3.4e-4932 ~ 1.1e4932

字节数

取值范围

关键字

浮点数采用IEEE 754标准存储,包括符号位、指数位和尾数位。

定长浮点型(Since C++23)

从 C++23 开始,引入了定长浮点型,如 float16_t, float32_t, float64_t 等,确保了数据类型的字节 数,保证了跨平台的兼容性 Reference

特殊值

- 正无穷、负无穷: ∞, -∞
- NaN(Not a Number): 0/0, ∞/∞ 等情况

```
std::numeric_limits<double>::infinity();
std::numeric_limits<double>::quiet_NaN();
std::numeric_limits<double>::signaling_NaN();
```

精度问题和舍入误差

浮点数的精度有限,可能会出现舍入误差。 在进行浮点数比较时,应该使用误差范围进行比较,而不是直接 比较两个浮点数的值。参见 std::numeric_limits<T>::epsilon()

上溢和下溢

C++ 中浮点数的上下溢是由于浮点数的表示范围有限,导致运算结果超出最大或最小可表示值,分别返回无穷 大或接近零的值。

void 类型

void 类型表示没有类型,通常用于函数返回值为空 或指针类型不确定的情况。 C 语言中,常用 void* 进行类型擦除

类型转换

- 显示类型转换:在需要时,可以使用强制类型转换 将一个类型转换为另一个类型
 - C-style cast: (type)expression
 - const_cast, reinterpret_cast
- 隐式类型转换:隐式类型转换是指编译器在不需要 显式指令的情况下自动进行的类型转换。常见的隐 式类型转换包括整型提升、浮点数提升、算术类型 ■ 转换等。

```
int a = 10:
float b = (float)a; // C-style cast
double b = static cast<double>(a); // C++ style cast
```

整形提升: char → int → long → long long 原则:

- 低精度类型转换为高精度类型,计算时运算符两边 的类型需要相等
- 长度小于 int 的整型提升为 int

C++ style cast: static_cast, dynamic_cast,浮点数提升: float → double → long double 原则:

- 低精度类型转换为高精度类型,计算时运算符两边 的类型需要相等
- 整形转化为浮点数

```
#include <iostream>
int main() {
    char a= 127;
    char b = 3;
    int c = a + b;
    std::cout << c << std::endl;
    return 0;
}</pre>
```

C++ style cast v.s. C-style cast

- static_cast 提供了编译期类型检查,避免了潜在的类型错误
- static_cast 不能用于无关类型之间的转换,如 int 转 char,这时可以使用 reinterpret_cast 进 行二进制数据的直接转换。可能因为内存对齐问题导致未定义行为
- C-style cast 会忽略任何检查

常量和字面量

■ 常量是指在程序运行过程中其值不会发生变化的量,使用 const 关键字进行声明,以保证其值不会被修改。该保护仅为编译期保护。

```
constexpr int x = 10;
constexpr int y = 10;
constexpr int sum = x + y; // 编译期计算

const int a = 10;
a = 20; // Error
```

■ 字面量(literal)是指在程序中直接出现的常量值,如 10, 3.14, 'a' 等。C++ 中的字面量包括整型、浮点型、字符型、字符串型、布尔型等。

```
int a = 10; // 整型字面量
float b = 3.14; // 浮点型字面量
char c = 'a'; // 字符型字面量
bool d = true; // 布尔型字面量
char e[] = "Hello, World!"; // 字符串型字面量
```

整形和浮点型字面量可以使用后缀 `u. l. f` 等进行修饰,以表示其类型。

Reference

字符串字面量

字符串字面量是由双引号括起来的字符序列,如 "Hello, World!"。字符串字面量是一个字符数组,以空字符 \0 结尾,表示字符串的结束。

- 常见的字符串编码方式有 ASCII 编码、UTF-8 编码、UTF-16 编码等。
- **转义字符**是指在字符串中以反斜杠 \ 开头的字符,用于表示一些特殊字符,如 \n 表示换行符, \t 表示制表符等。要表示反斜杠本身,需要使用 \\ 。
- **原始字符串字面量**是指以 R"()" 包裹的字符串字面量,不会对转义字符进行转义,如 R"(Hello, \n World!)"。

为了支持 Unicode 字符串,C++11 引入了 u8 前缀,表示字符串使用 UTF-8 编码,如 u8"你好,世界!",同时 wchar_t 类型也被引入,用于表示宽字符。

User defined literals(Since C++11)

C++11 开始,支持用户自定义字面量。类似一个 consteval 的函数。

Syntax

指针和引用

指针(pointer)是一个存储变量地址的变量,可以通过指针访问变量的值。引用(reference)是一个变量的别名,可以通过引用访问变量的值。 std::nullptr_t 是一个空指针类型,用于表示空指针。

```
int a = 10;
int* p = &a; // 指针
int* r = a; // 引用
int* p = nullptr; // 空指针
*p = 11; // 通过指针访问变量
r = 12; // 通过引用访问变量
```

指针算术:

■ 指针加减整数:指针加减整数会根据指针类型的大小进行移动 sizeof(type)

指针和引用的区别:

- 指针可以被重新赋值,引用不能(指向新的对象)
- 指针可以为空,引用不能

复合数据类型 数组

数组是一种存储相同类型数据的集合,数组的大小在声明时就确定,且不可改变。数组的元素可以通过下标访问,下标从 0 开始。

```
int arr[5] = {1, 2, 3, 4, 5}; // 初始化数组
int arr[5]; // 定义数组
int arr[] = {1, 2, 3, 4, 5}; // 自动计算数组大小
int arr[5] = {1, 2}; // 部分初始化, 结果为 {1, 2, 0, 0, 0}

arr[0] = 10; // 访问并修改数组元素
*(arr + 1) = 20; // 指针访问数组元素
```

Reference:

- https://en.cppreference.com/w/c/language/array
- https://en.cppreference.com/w/c/language/array_initialization

数组退化

在函数参数传递中,数组会退化为指针,因此在函数参数中声明数组时,实际上是声明了一个指针。

```
# include <iostream>
// void foo(int *args)
// void foo(int args[5])
void foo(int args[]) {
  // args 退化为指针
  static assert(sizeof(args) = sizeof(int*), "Error");
  std::cout << sizeof(args) << std::endl: // 8</pre>
int main() {
  int arr[5] = \{1, 2, 3, 4, 5\};
  std::cout << sizeof(arr) << std::endl; // 20
  foo(arr);
  return 0;
```

结构体(C struct)

结构体是一种用户自定义的数据类型,用于存储不同类型的数据。结构体的成员可以是基本数据类型、数组、 指针、结构体等。

```
struct Student {
    int id;
    char name[20];
    float score;
};
struct Class {
    int id;
    Student students[30];
};
struct Student s = {1, "Alice", 90.5};
struct Student t = {.id = 1, .name = "Alice", .score = 90.5};
struct Class c = {1, {s, t}};
// access struct member
std::cout << s.id << s.student[0].name << std::endl;</pre>
```

Union(联合体或共用体)

联合体是一种特殊的结构体,所有成员共用同一块内存空间,只能同时存储一个成员的值。Union 的大小等于最大成员的大小。

```
Union U {
    int a;
    double b;
};

size_t size = sizeof(U); // 8

U u;
u.a = 10;
std::cout << u.a << std::endl; // 10
u.b = 3.14;
std::cout << u.b << std::endl; // 3.14</pre>
```

C++17 引入了 std::variant 类型,用于替代 Union,提供了更好的类型安全性和异常处理。

enum(枚举)

枚举是一种用户自定义的数据类型,用于定义一组具名整型常量。枚举类型的值可以通过枚举常量名访问。本质上枚举类型是整型类型,可以通过 static_cast 进行转换。

```
#include <iostream>
int main() {
  enum Color { RED, GREEN, BLUE };

enum Weather : long { SUNNY = 10, RAINY = 20, CLOUDY = 30 };

enum Unknow : int;

Color c = RED;
Weather w = SUNNY;

std::cout << c << '\n' << w << std::endl;
}</pre>
```

由于 C 语言中的枚举类型是全局的,容易引起命名冲突,C++11 引入了枚举类(enum class),用于解决这个问题。

Reference

Control Flow

选择语句

if 语句

```
if (condition) {
    // code block
} else if (condition) {
    // code block
} else {
    // code block
}
```

switch 语句

```
switch (expression) {
    case constant1:
        // code block
        [[fallthrough]];
    case constant2:
    case constant3:
        // code block
        break;
    default:
        // code block
}
```

循环语句 while 语句

```
while (condition)
  // statement

do {
    // code block
} while (condition);
```

for 语句

```
for (int i = 0; i < 10; i++)
  // statement

int b = 10;
for (; b > 0; b--)
  // statement
```

跳转语句

■ break: 跳出当前循环

■ continue: 跳过当前循环

■ return: 返回函数值给调用者

■ goto: 无条件跳转到标签处

```
void foo() {
 for (int i = 0; i < 10; i++) {
     if (i = 5) {
         break;
     if (i = 3) {
         continue;
     if (i = 7) {
         goto end;
  end:
  return;
```

Expression

表达式是由**操作数**和**运算符** (operator) 组成的计算式,可以计算出一个值。表达式可以是常量、变量、函数调用、运算符等。

```
int a = 10;
int b = 20 + a;
int c = a + b * 2;
if(a > 10 \& b < 20 || c = 30);
a++;
--b;
a += 2;
int d = a > b ? a : b;
int d, e = 2;
a = b = c = d = e = 0;
int ary[10];
ary[4] = 10;
int** ptr = &ary;
(*ptr)[4] = 10;
```

Reference

Q&A