深蓝学院
shenlanxueyuan.com

C++ 基础

第 2 章：对象与基本类型

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

从初始化 / 赋值语句谈起

```
int x = 10;   x = 20;
```

- 初始化 / 赋值语句是程序中最基本的操作，其功能是将某个值与一个对象关联起来
  - 值：字面值、对象（变量或常量）所表示的值……
  - 标识符：变量、常量、引用……
  - 初始化基本操作：
    - 在内存中开辟空间，保存相应的数值
    - 在编译器中构造符号表，将标识符与相关内存空间关联起来
  - 值与对象都有类型
  - 初始化 / 赋值可能涉及到类型转换

类型详述

- 类型是一个编译期概念，可执行文件中不存在类型的概念
- C++ 是强类型语言
- 引入类型是为了更好地描述程序，防止误用
- 类型描述了：
  - 存储所需要的尺寸 (sizeof，标准并没有严格限制)
  - 取值空间 (std::numeric_limits，超过范围可能产生溢出)
  - 对齐信息（alignof）

```
std::cout << alignof(int) << std::endl;
```

  - 可以执行的操作

```
#include <iostream>

// #pragma pack (1)
struct Str
{
    // 8000
    char b;

    // 8004~8007
    int x;
};

int main() {
    std::cout << sizeof(Str) << std::endl;
}
```

```
Program returned: 0
Program stdout
8
```

```
#include <iostream>

#pragma pack (1)
struct Str
{
    // 8000
    char b;

    // 8004~8007
    int x;
};

int main() {
    std::cout << sizeof(Str) << std::endl;
}
```

```
Program returned: 0
Program stdout
5
```

类型详述

- 类型可以划分为基本类型与复杂类型
  - 基本（内建）类型： C++ 语言中所支持的类型
    - 数值类型
      - 字符类型（char, wchar_t, char16_t, char32_t）
      - 整数类型
        - 带符号整数类型： short, int, long, long long
        - 无符号整数类型： unsigned + 带符号整数类型
      - 浮点类型
        - float, double, long double
    - void
  - 复杂类型：由基本类型组合、变种所产生的类型，可能是标准库引入，或自定义类型

char：8位，一个字节,通常用来表示ASCII

char32_t,可以表示一些复杂的字符。例如汉字，希腊字母等

```
char32t wc[]= U"水";
```

类型详述

```
int main()
{
    char ch1;
    unsigned char ch2;

    signed char ch3;
}
```

- 与类型相关的标准未定义部分
  - char 是否有符号 根据编译器不同和不同。有的编译器认为有符号，有的编译器认为无符号
  - 整数中内存中的保存方式：大端 小端

  - 每种类型的大小（间接影响取值范围）
    - C++11 中引入了固定尺寸的整数类型，如 int32_t

unsigned == unsigned int

类型详述—字面值及其类型

- 字面值：在程序中直接表示为一个具体数值或字符串的值
- 每个字面值都有其类型
  - 整数字面值： 20 （十进制）， 024 （八进制）， 0x14 （十六进制） -- int 型
  - 浮点数： 1.3, 1e8 – double 型
  - 字符字面值： 'c', '\n', '\x4d' – char 型
  - 字符串字面值： "Hello" – char[6] 型 字符串中会自动引入 \0 ,来代表字符串结束
  - 布尔字面值： true, false – bool 型
  - 指针字面值： nullptr – nullptr_t 型

类型详述—字面值及其类型

- 可以为字面值引入前缀或后缀以改变其类型
  - 1.3 （double） -- 1.3f （float）
  - 2 （int） -- 2ULL (unsigned long long)
- 可以引入自定义后缀来修改字面值类型 用途：可以定义_s，做物理的时间的概念。返回一个时间的struct

```cpp
#include <iostream>

int operator ""_ddd(long double x)
{
    return (int)x * 2;
}

int main() {
    int x = 3.14_ddd;
    std::cout << x << '\n';
}
```

```
Program returned: 0
Program stdout
6
```

类型详述—变量及其类型

- 变量：对应了一段存储空间，可以改变其中内容
- 变量的类型在其首次声明（定义）时指定：
  - int x：定义一个变量 x ，其类型为 int
  - 变量声明与定义的区别： extern 前缀
- 变量的初始化与赋值
  - 初始化：在构造变量之初为其赋予的初始值
    - 缺省初始化 `int x;`

如果我们全局缺省初始化一个变量，那么这个变量会被初始化为0.=》静态变量
如果我们在函数内部缺省初始化一个变量。那么这个变量是未定义的。可能是0，也可能是上一次操作遗留下来的数字。因为c++是一个注重效率的语言。如果我们对一个缺省的变量初始化，因为我们不知道这个函数会被调用多少次。那么会消耗额外的cpu，资源。=》非静态变量

    - 直接 / 拷贝初始化

`int x(10);` `int z{10};` `int x = 10;`

    - 其它初始化
  - 赋值：修改变量所保存的数值

类型详述——（隐式）类型转换

- 为变量赋值时可能涉及到类型转换
  - bool 与 整数之间的转换
  - 浮点数与整数之间的类型转换
- 隐式类型转换不只发生在赋值时
  - if 判断
  - 数值比较
    - 无符号数据与带符号数据之间的比较

在无符号数和有符号数做比较时，会将带符号数转化为无符号数。

    - std::cmp_XXX （C++ 20）

```
int main()
{
    int x = -1;
    unsigned int y = 3;
    std::cout << (x < y) << std::endl;
}
```

输出0

复合类型：从指针到引用

- 指针：一种间接类型

- 特点
  - 可以“指向”不同的对象
  - 具有相同的尺寸
- 相关操作
  - & – 取地址操作符
  - * – 解引用操作符

复合类型：从指针到引用

- 指针的定义
  - int* p = &val;
  - int* p = nullptr;
- 关于 nullptr
  - 一个特殊的对象（类型为 nullptr_t），表示空指针
  - 类似于 C 中的 NULL，但更加安全
- 指针与 bool 的隐式转换：非空指针可以转换为 true；空指针可以转换为 false
- 指针的主要操作：解引用；增加、减少；判等
- void* 指针
  - 没有记录对象的尺寸信息，可以保存任意地址
  - 支持判等操作

```cpp
#include <iostream>

int main() {
    int *p = 0;
    std::cout << (*p) << std::endl;
}
```

在我们不知道要付给指针什么初始值的时候，可以先赋予0.这样子在解引用的时候就会报错。不过由于c++的自动初始化，造成不必要的麻烦。

```cpp
#include <iostream>

int main() {
    int *p = nullptr;
}
```

```cpp
int *p = nullptr;

if (p) {

} else {

}
```

进行了与bool的隐式转换

```cpp
void fun(void* param) {

}
```

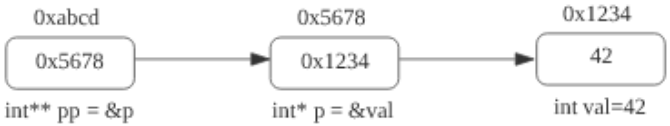
我们只关注它是一个指针。并不关注到底是什么类型的指针。因为指针的尺寸是一样的。所以才可以这么保存

```cpp
#include <iostream>

void fun(void* param) {
    std::cout << (param + 1) << std::endl;
}

int main() {
    int x = 42;
    int* r = &x;

    std::cout << r << std::endl;
    std::cout << r + 1 << std::endl;
    fun(r);
}
```

```
Could not execute the program
Compiler returned: 1
Compiler stderr
<source>:4:25: error: arithmetic on a pointer to void
    std::cout << (param + 1) << std::endl;
                 ~~~~~ ^
1 error generated.
```

复合类型：从指针到引用

- 指针的指针



- 指针 V.S. 对象 我们有的时候还可以要求某些struct不支持复制，是独占的。此时就只能使用引用或指针来传入function
  - 指针复制成本低，读写成本高 因为需要解引用。因此读写成本高
    指针是一种间接引用。间接引用可以减少传输的成本。无论原来的对象是几个字节，指针都是8个字节。可以减少传输成本
- 指针的问题
  - 可以为空
  - 地址信息可能非法
  - 解决方案：引用

复合类型：从指针到引用

- 引用　不可以构造引用的引用。因为引用不是一个对象
  - int& ref = val;
  - 是对象的别名，不能绑定字面值
  - 构造时绑定对象，在其生命周期内不能绑定其它对象（赋值操作会改变对象内容）
  - 不存在空引用，但可能存在非法引用——总的来说比指针安全
  - 属于编译期概念，在底层还是通过指针实现

    引用其实就是为了解决指针可能出现地址为空的现象。在底层其实还是通过指针实现。
- 指针的引用
  - 指针是对象，因此可以定义引用
  - int* p = &val; int* & ref = p;
  - 类型信息从右向左解析

非法引用：引用指向了一个已经被销毁的对象。下图中的x在fun()之后就已经不存在了。

```
int& fun()
{
    int x;
    return x;
}

int main()
{
    int& res = fun();
}
```

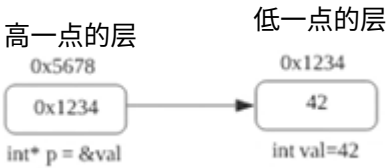
常量类型与常量表达式

```
int main()
{
    // const int*  -x-> int*
    const int x = 3;
    int* ptr = &x;
}
```

这种情况是不可以的

- 常量与变量相对，表示不可修改的对象
  - 使用 const 声明常量对象 const概念并没有底层硬件支持，它是由编译器实现的
  - 是编译期概念，编译器利用其
    - 防止非法操作
    - 优化程序逻辑 int const y = 3; 和 const int y = 3;是等价的

限制指针本身不能修改



指针内容不能修改

- 常量指针与顶层常量（top-level const）
  - const int* p; = &x。 x的值不能被改变。
  - int* const p; = &x; p 指针不能被修改,只能指向x。
  - const int* const p; 以*为分界线，如果const在右侧，表示指针本身不能改变。 如果const在左侧，表示指向的内容不能改变。
  - 常量指针可指向变量

```
int main()
{
    int x = 4;
    const int* ptr = &x;
}
```

const int* 在初始化的时候进行了隐式的类型转换
int* --> const int*
这种转换是合理的。因为这种转换是增加了限制。

常量类型与常量表达式

```cpp
#include <type_traits>

int main()
{
    constexpr const int* ptr = nullptr;
    std::cout << std::is_same_v<decltype(ptr), const int* const> << std::
}
```

- 常量引用（也可绑定变量）
  - const int&
  - 可读但不可写
  - 主要用于函数形参
  - 可以绑定字面值

如果是int类型，建议直接写void fun(int param)，而不是void fun(const int& param)。因为原来的int可能只有4个字节。而使用引用就变成了拷贝地址。可能就变成8个字节。还使内存消耗增加了。然后每次访问param，都要从地址读取，还减慢了读取的速度。
因为传的是形参，所以其实也不需要写const。他不会改变主函数中的param的值

```cpp
int main()
{
    int x = 3;
    const int& ref = 3;
}
```

常量引用可以绑定字面值。
引用不可以绑定字面值
int x = 3;
int& ref = 3; // error

```cpp
void fun(const int& param)
{
}

int main()
{
    int x = 3;
    fun(x);
    fun(3);
}
```

因为我们希望上面的fun(3)这种写法的程序work。因此c++会允许常量引用绑定字面值。

- 常量表达式 （从 C++11 开始）
  - 使用 constexpr 声明
  - 声明的是编译期常量
  - 编译器可以利用其进行优化
  - 常量表达式指针： constexpr 位于 * 左侧，但表示指针是常量表达式

expression

编译器常量
constexpr int y = 3;
y的类型还是 const int

```
int x = 3;
constexpr const int* ptr = nullptr; // 表示ptr是一个在编译期确定的常量
ptr -> const int* const
constexpr 并不属于类型声明的一部分。它只是修饰了ptr.
```

类型别名与类型的自动推导

- 可以为类型引入别名，从而引入特殊的含义或便于使用（如： size_t ）

无符号整形，可以表示任意尺寸的对象

- 两种引入类型别名的方式
  - typedef int MyInt; 原始的名称写在前面，新的名称写在后面
  - using MyInt = int; （从 C++11 开始） 新的名称写在开头，原来的名称写在结尾
- 使用 using 引入类型别名更好
  - typedef char MyCharArr[4];
  - using MyCharArr = char[4];
- 类型别名与指针、引用的关系
  - 应将指针类型别名视为一个整体，在此基础上引入常量表示指针为常量的类型
  - 不能通过类型别名构造引用的引用

```
using IntPtr = int*;
```

```
using IntPtr = int*;

int main()
{
    int x = 3;
    const IntPtr ptr = &x;
    int y = 0;
    ptr = &y;
}
```

上面的程序会报错。因为const IntPtr会被视为指针为常量。不能把y给ptr。相当于如下定义

```
int* const ptr = &x;
```

类型别名与类型的自动推导

c++是强类型语言。而python是一个弱类型语言

- 类型的自动推导
  - 从 C++11 开始，可以通过初始化表达式自动推导对象类型
  - 自动推导类型并不意味着弱化类型，对象还是强类型
  - 自动推导的几种常见形式
    - auto：最常用的形式，但会产生类型退化
    - const auto / constexpr auto：推导出的是常量 / 常量表达式类型
    - auto&：推导出引用类型，并且避免类型退化
    - decltype(exp)：返回 exp 表达式的类型（左值加引用）
    - decltype(val)：返回 val 的类型
    - decltype(auto)：从 c++14 开始支持，简化 decltype 使用
    - concept auto：从 C++20 开始支持，表示一系列类型（ std::integral auto x = 3; ）

```
decltype三部曲
1. 返回表达式类型
decltype(3.5 + 15l)->double
2. 表达式是左值，会加引用
decltype(*ptr) -> int&
decltype((x)) -> int& // (x)不是变量名称，且可左值
3. decltype 返回变量名称。
decltype(x) -> int
decltype(ptr) -> int*
*ptr不是变量名称，它是一个表达式
```

```
int main()
{
    int x1 = 3;
    int& ref = x1;

    auto ref2 = ref;

    std::cout << std::is_same_v<decltype(ref2), int> << std::endl;
}
```

输出是1

```
int x;
x = 3; // x作为左值
int y = x; // x作为右值
```

```
典型的类型退化是引用。
int x = 3;
int& ref = x;
auto ref2 = ref; // ref2的类型变成了int，不再是int&
int y = ref; // 此时（在右值引用的时候）ref会从int& 退化为int
// const int& 也会退化为int
// 数组在右值引用的时候会退化为指针
```

declaration　　decltype不会产生类型退化

```
int x = 3;
int& y1 = x;
decltype(y1) y2 = y1;
```

```
decltype(3.5 + 15l) x = 3.5 + 15l;
```

```
const int x = 3;
auto& y = x; // 当编译器看到auto& 的时候，就不会对x进行退化了，会将x继续视为const int
```

```
int main()
{
    const int x = 3;
    auto& y = x;
    std::cout << std::is_same_v<decltype(y), const int&> << std::endl;
}
```

```
ptr -> int*
*ptr -> int (左值)
decltype(*ptr) -> int&
```

```
int main()
{
    int x[3] = {1,2,3};
    auto x1 = x;
    std::cout << std::is_same_v<decltype(x1), int*> << std::endl;
}
```

```
decltype(3.5 + 15l) x = 3.5 + 15l;
decltype(auto) x = 3.5 + 15l;
```

```
int x[3] = {1,2,3};
auto& x1 = x;
std::cout << std::is_same_v<decltype(x1), int(&)[3]> << std::endl;
```

域与对象的生命周期

- 域 (scope) 表示了程序中的一部分，其中的名称有唯一的含义
- 全局域（ global scope ）：程序最外围的域，其中定义的是全局对象
- 块域（ block scope ），使用大括号所限定的域，其中定义的是局部对象
- 还存在其它的域：类域，名字空间域……
- 域可以嵌套，嵌套域中定义的名称可以隐藏外部域中定义的名称
- 对象的生命周期起始于被初始化的时刻，终止于被销毁的时刻
- 通常来说
  - 全局对象的生命周期是整个程序的运行期间
  - 局部对象生命周期起源于对象的初始化位置，终止于所在域被执行完成

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening