加油！当自己的实力不能满足自己的目标时，**就静下心去学习！**  
总结 思考 反思

@[toc]

# 内存分区模型

C++程序在执行时，将内存大方向划分为**4个区域**

* 代码区：存放函数体的二进制代码，由操作系统进行管理的
* 全局区：存放全局变量和静态变量以及常量
* 栈区：由编译器自动分配释放, 存放函数的参数值,局部变量等
* 堆区：由程序员分配和释放,若程序员不释放,程序结束时由操作系统回收

**内存四区意义：**

* 不同区域存放的数据，赋予不同的生命周期, 给我们更大的灵活编程

## 1、程序运行前

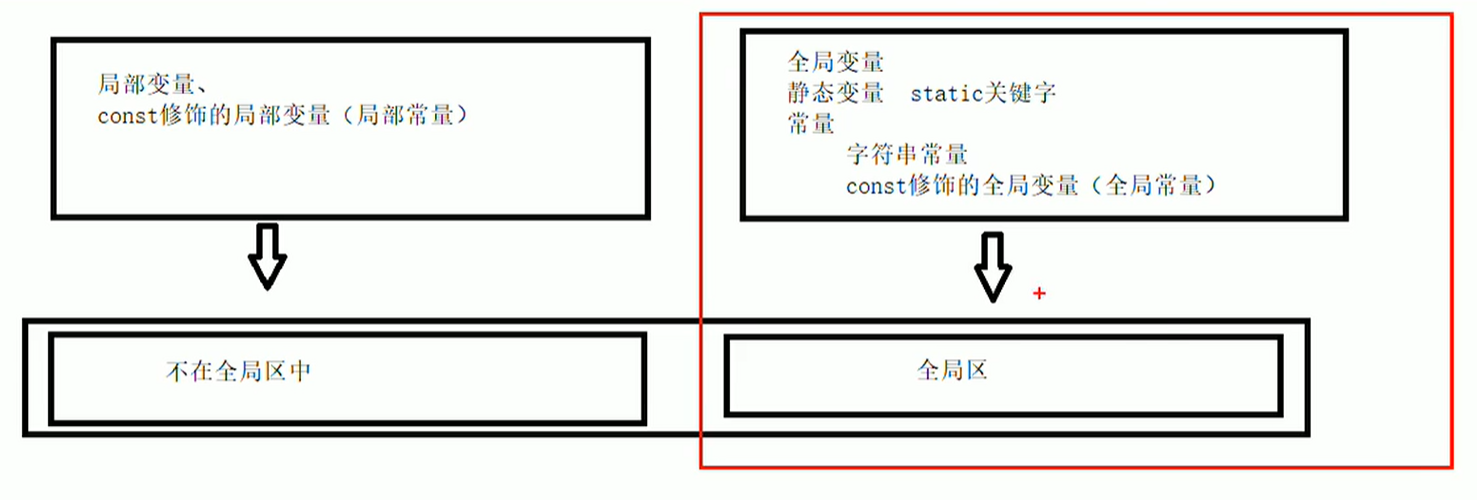
* 在程序编译后，生成了exe可执行程序，**未执行该程序前**分为两个区域

**代码区：**

* 存放 CPU 执行的机器指令
* 代码区是**共享**的，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可
* 代码区是**只读**的，使其只读的原因是防止程序意外地修改了它的指令

**全局区：**

* 全局变量和静态变量存放在此.
* 全局区还包含了常量区, 字符串常量和其他常量也存放在此.
* 该区域的数据在程序结束后由操作系统释放.

  
总结：

* C++中在程序运行前分为全局区和代码区
* 代码区特点是共享和只读
* 全局区中存放全局变量、静态变量、常量
* 常量区中存放 const修饰的全局常量 和 字符串常量

## 2、程序运行后

**栈区**

* 由编译器自动分配释放, 存放函数的参数值,局部变量等
* 注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

**示例：**

int \* func()  
{  
 int a = 10;  
 return &a;  
}  
  
int main() {  
  
 int \*p = func();  
  
 cout << \*p << endl; //正确，因为编译器会有一次保留  
 cout << \*p << endl; // 错误，不能返回局部变量的地址  
  
 system("pause");  
  
 return 0;  
}

**堆区：**

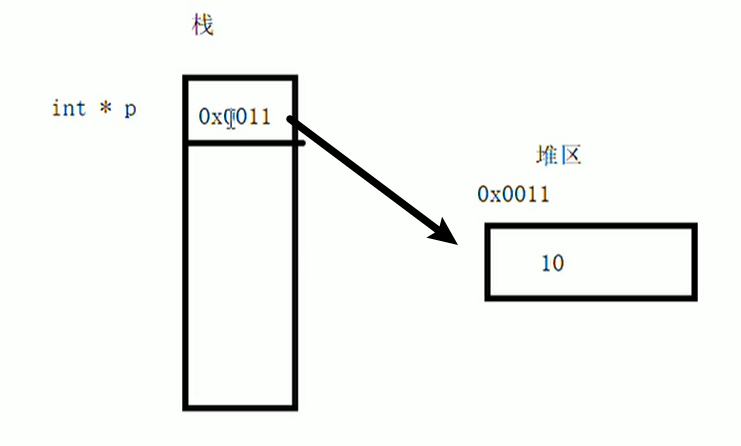
* 由程序员分配释放,若程序员不释放,程序结束时由操作系统回收
* 在C++中主要利用new在堆区开辟内存

**示例：**

int\* func()  
{  
 int\* a = new int(10);  
 return a;  
}  
  
int main() {  
  
 int \*p = func();  
  
 cout << \*p << endl; // 这里的\*P 为解引用操作 输出正确  
 cout << \*p << endl; // 输出正确  
   
 system("pause");  
  
 return 0;  
}

**总结：**

* 堆区数据由程序员管理开辟和释放
* 堆区数据利用new关键字进行开辟内存



## 3、new操作符

* C++中利用new操作符在堆区开辟数据，那么如何进行释放呢？
* 堆区开辟的数据，由程序员手动开辟，手动释放，释放利用操作符 delete
* 语法：new 数据类型
* 利用new创建的数据，会返回该数据对应的类型的指针

**示例1： 基本语法**

int\* func()  
{  
 int\* a = new int(10);  
 return a;  
}  
  
int main() {  
  
 int \*p = func();  
  
 cout << \*p << endl;  
 cout << \*p << endl;  
  
 //利用delete释放堆区数据  
 delete p;  
  
 //cout << \*p << endl; //报错，释放的空间不可访问  
  
 system("pause");  
  
 return 0;  
}

**示例2：开辟数组**

//堆区开辟数组  
int main() {  
  
 int\* arr = new int[10];  
  
 for (int i = 0; i < 10; i++)  
 {  
 arr[i] = i + 100;  
 }  
  
 for (int i = 0; i < 10; i++)  
 {  
 cout << arr[i] << endl;  
 }  
   
 //释放数组 delete 后加 []，否则释放的是第一个数据  
 delete[] arr;  
  
 system("pause");  
  
 return 0;  
}

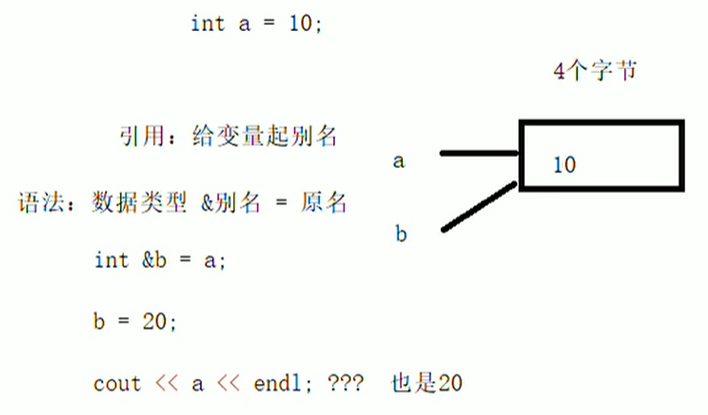
# 引用

## 引用的基本使用

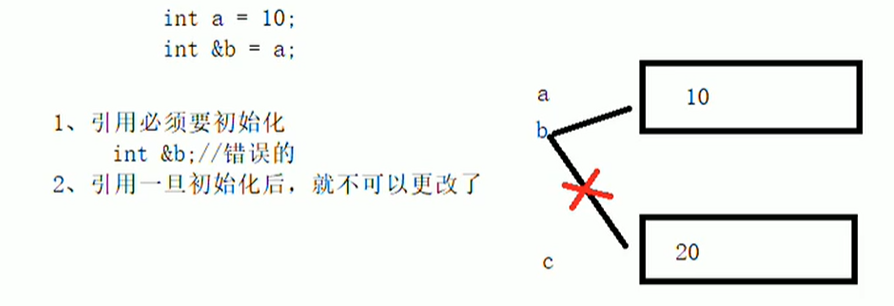
* **作用**：给变量起别名
* **语法：** 数据类型 &别名 = 原名

**示例：**

int main() {  
  
 int a = 10;  
 int &b = a;  
  
 cout << "a = " << a << endl;  
 cout << "b = " << b << endl;  
  
 b = 100;  
  
 cout << "a = " << a << endl;  
 cout << "b = " << b << endl;  
  
 system("pause");  
  
 return 0;  
}



## 引用注意事项

* 引用必须初始化
* 引用在初始化后，不可以改变  
  

示例：

int main() {  
  
 int a = 10;  
 int b = 20;  
 //int &c; //错误，引用必须初始化  
 int &c = a; //一旦初始化后，就不可以更改  
 c = b; //这是赋值操作，不是更改引用  
  
 cout << "a = " << a << endl; //20  
 cout << "b = " << b << endl; //20  
 cout << "c = " << c << endl; //20  
  
 system("pause");  
  
 return 0;  
}

## 引用做函数参数（注意）

* **作用** : 函数传参时，可以利用引用的技术让形参修饰实参
* **优点** : 可以简化指针修改实参

**示例：**

//1. 值传递  
void mySwap01(int a, int b) {  
 int temp = a;  
 a = b;  
 b = temp;  
}  
  
//2. 地址传递  
void mySwap02(int\* a, int\* b) {  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
//3. 引用传递  
void mySwap03(int& a, int& b) {  
 int temp = a;  
 a = b;  
 b = temp;  
}  
  
int main() {  
  
 int a = 10;  
 int b = 20;  
  
 mySwap01(a, b); // 值传递，形参不会修饰实参  
 cout << "a:" << a << " b:" << b << endl;  
  
 mySwap02(&a, &b); // 地址传递，形参会修饰实参  
 cout << "a:" << a << " b:" << b << endl;  
  
 mySwap03(a, b); // 采用引用传递，形参会修饰实参数  
 cout << "a:" << a << " b:" << b << endl;  
  
 system("pause");  
  
 return 0;  
}

总结：  
1、通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单  
2、原因： 采取别名的方式

## 引用做函数返回值

* **作用**：引用是可以作为函数的返回值存在的
* **注意**：**不要返回局部变量引用** (类似于返回局部变量的地址)
* **用法**：函数调用作为左值

**示例：**

//返回局部变量引用  
int& test01() {  
 int a = 10; //局部变量  
 return a;  
}  
  
//返回静态变量引用  
int& test02() {  
 static int a = 20;  
 return a;  
}  
  
int main() {  
  
 //不能返回局部变量的引用  
 int& ref = test01();  
 cout << "ref = " << ref << endl;  
 cout << "ref = " << ref << endl;  
  
 //如果函数做左值，那么必须返回引用  
 int& ref2 = test02();  
 cout << "ref2 = " << ref2 << endl;  
 cout << "ref2 = " << ref2 << endl;  
  
 test02() = 1000;  
  
 cout << "ref2 = " << ref2 << endl;  
 cout << "ref2 = " << ref2 << endl;  
  
 system("pause");  
  
 return 0;  
}

## 引用的本质

本质：**引用的本质在c++内部实现是一个**指针常量**.**

讲解示例：

//发现是引用，转换为 int\* const ref = &a;  
void func(int& ref){  
 ref = 100; // ref是引用，转换为\*ref = 100  
}  
int main(){  
 int a = 10;  
   
 //自动转换为 int\* const ref = &a; 指针常量是指针指向不可改，也说明为什么引用不可更改  
 int& ref = a;   
 ref = 20; //内部发现ref是引用，自动帮我们转换为: \*ref = 20;  
   
 cout << "a:" << a << endl;  
 cout << "ref:" << ref << endl;  
   
 func(a);  
 return 0;  
}

结论：C++推荐用引用技术，因为语法方便，引用本质是指针常量，但是所有的指针操作编译器都帮我们做了

指针常量、常量指针

[跳转链接](https://blog.csdn.net/QuantumYou/article/details/118887902?spm=1001.2014.3001.5502)





## 常量引用

**作用** : 常量引用主要用来修饰形参，防止误操作

* 在函数形参列表中，可以加const修饰形参，防止形参改变实参

**示例：**

//引用使用的场景，通常用来修饰形参  
void showValue(const int& v) {  
 //v += 10;  
 cout << v << endl;  
}  
  
int main() {  
  
 //int& ref = 10; 引用本身需要一个合法的内存空间，因此这行错误  
 //加入const就可以了，编译器优化代码，int temp = 10; const int& ref = temp;  
 const int& ref = 10;  
  
 //ref = 100; //加入const后不可以修改变量  
 cout << ref << endl;  
  
 //函数中利用常量引用防止误操作修改实参  
 int a = 10;  
 showValue(a);  
  
 system("pause");  
  
 return 0;  
}

注意：函数中利用常量引用防止误操作修改实参