Modern C++ Camp

现代C++系统研发骨干特训营

李建忠 Boolan

课程目标

Boolan
高端IT咨询与教育平台

模块一、C++类型系统与设施

模块二、C++面向对象编程

模块三、C++内存管理

模块四、模板机制

模块五、泛型编程与STL

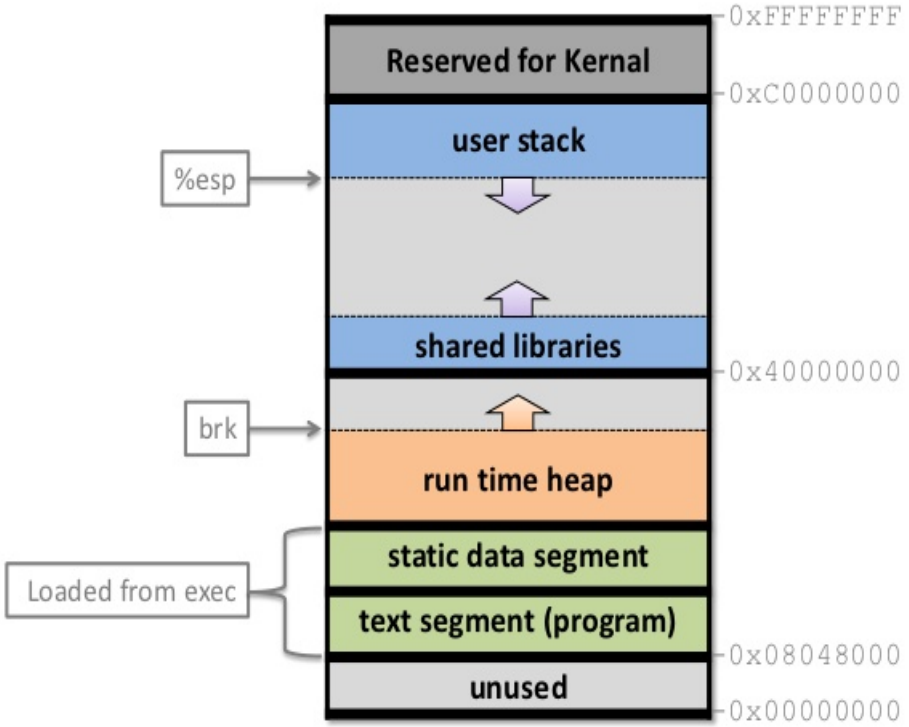
Boolan
高端IT咨询与教育平台

现代C++系统研发骨干特训营

模块一、C++类型系统与设施

数据存储

- 程序数据段
  - 静态（全局）数据区：全局变量、静态变量
  - 堆内存：程序员手动分配、手动释放
  - 栈内存：编译器自动分配、自动释放
  - 常量区：编译时大小、值确定不可修改
- 程序代码段
  - 函数体



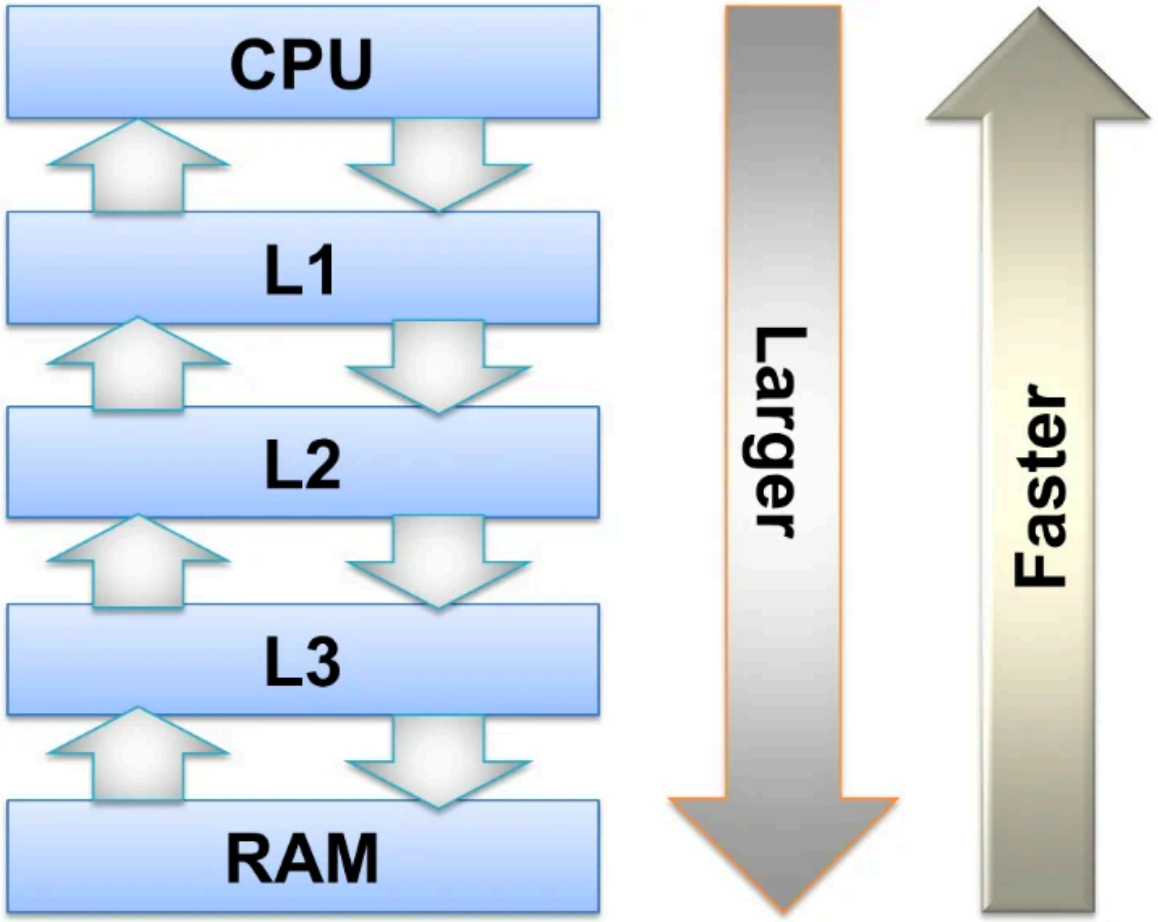
Stack 栈内存

- 栈内存属于执行期函数，编译时大小确定
- 函数执行时，栈空间自动分配
- 函数结束时，栈空间自动销毁
- 栈上对象线性分配，连续排列，没有内存碎片效应
- 栈内存具有函数本地性，不存在多线程竞争
- 栈内存有大小限制，可能会溢出，例如Linux默认为8MB，Windows默认为1MB
- 栈内存使用对象或引用直接使用，管理复杂度低

Heap 堆内存

- 堆内存属于具有全局共享特点，大小动态变化
- 对象分配时，手动分配堆内存（malloc/new）
- 对象释放时，手动释放堆内存（delete/free）
- 堆上对象链式分配，非连续排列
- 堆内存全局共享，存在多线程竞争可能性
- 堆内存大小没有栈内存严格限制，与机器内存总量和进程寻址空间相关
- 堆内存使用指针间接访问，管理复杂度高
- 堆内存有很高灵活性，虽性能较差，但可通过相关设施和编程技巧精细控制，从而获得改善。

寄存器+多级缓存+主存的多级存储架构



示例：Intel i7的存储架构（主频3.4GHz）

名称	大小	延迟
寄存器	~1KB	—
L0（微码）缓存	6KB	—
L1 缓存	32KB数据+32KB指令	4时钟周期（数据）
L2 缓存	256KB	11时钟周期（ ~4纳秒）
L3 缓存（4核共享）	8MB	39时钟周期（ ~10纳秒）
主存	32GB(最大；~$3/GB)	107时钟周期
固态盘	256GB(典型；~$0.2/GB)	~30微妙
硬盘	2TB(典型；~$0.05/GB)	~15毫秒
磁带	15TB(典型；~$0.01/GB)	无随机访问能力

性能指南——堆-栈内存

- 栈内存分配快，布局连续，缓存友好，释放快
- 如果生存周期短，拷贝较少（传参、返回值），栈内存性能更好
- 堆内存有很高灵活性，但性能较差
- 堆内存在长运行程序有内存碎片效应，小块空闲内存得不到重用
- 堆分配需要寻找合适大小内存块，会花费更多时间
- 堆空间碎片化，容易降低缓存效率
- 编译器较难优化使用指针的代码
- 使用者需要确保申请释放成对，避免内存泄漏导致堆内存耗尽
- 使用者需要确保内存释放后不能访问（悬浮指针）
- 可以通过RAII 和指针移动操作避免拷贝代价。

值语义与引用语义

- 对内置类型和用户自定义类型提供同等支持。不存在特权类型或限定（其他语言则不，这是C++的独特性之一）。

- 任何类型的实例都同时具有值语义 和 引用语义：
  - 值语义：对象以值的方式直接存储，传参、返回值、拷贝等。
  - 引用语义：对象以指针或引用的方式间接存储，参数、返回值、拷贝传递的是指针或引用。

- 值语义有很多好处：没有悬浮指针/引用，没有昂贵的释放操作，没有内存泄漏，没有数据竞争…… 但是值语义大对象拷贝代价高昂，不能支持虚函数多态，不能维持对象全局唯一性……

变量的生命周期

- 全局变量，函数中的静态变量，类静态数据成员
  - 程序启动后加载，程序结束释放。
- 局部变量、自动对象（栈）：
  - 自声明开始，到声明语句所在块结尾 }释放
- 堆变量，自由存储对象
  - new开始，delete结束
- 临时对象
  - 和表达式周期一致，通常类似自动对象（绑定引用除外）
- 线程局部对象, thread_local
  - 随线程创建而创建，随线程结束而释放。

变量的初始化

- 统一初始化：int a1{100}; int a2={100};
- 赋值初始化：int a3=100;
- 构造初始化：int a4(100);

- 大多数情况推荐使用统一初始化，又叫列表初始化，特别是对象、容器；对于数值，可防止隐式窄化转型。空列表{}使用默认值初始化。
- 基本数值类型，以及auto自动推断类型声明，可以继续使用赋值初始化（除非需要避免数值窄化转型）。

指针与引用

- 辨析指针、引用所指向对象的存储位置、生命周期、以及是否拥有对象所有权。
- 尽可能避免以原始指针（T*）或引用（T&）来传递所有权（使用智能指针传递所有权）。默认情况下，指针不传递所有权。
- 以裸指针的形式传递的对象，假定由调用方所有，其生命周期也由调用方负责。
- 不涉及生命周期的函数应当接受裸指针或引用。
- 当不会改变被指代的对象时，引用通常比指针更好。

指针是万恶之源：内存错误的罪与罚

- 所有权不清晰（谁分配，谁释放？）
- 对象类型不清晰（栈对象、堆对象、数组对象、资源句柄？）
- 错误百出的指针
  - 内存泄漏——忘记delete之前new的内存
  - 悬浮指针—— 使用已释放内存（读取、或写入）、返回栈对象地址
  - 重复删除—— 对已经删除过的对象，进行二次删除
  - 删除非堆对象指针——对栈对象、全局/静态对象地址进行删除
  - 分配与删除错误匹配—— new和free搭配，malloc和delete搭配，new[]和delete搭配，new和delete[] 搭配
  - 使用空指针
  - 使用失效引用

基于对象编程

- 数据成员（字段） + 函数成员
- 对象有什么？
  - 实例成员与this指针
  - 静态成员
- 对象在哪里？——空间分析
  - 基本类型成员
  - 内嵌对象成员
  - 内嵌指针成员
- 操作符重载

C++ 对象模型基础

- C++对象内存布局
  - 按照实例数据成员声明顺序从上到下排列（与C语言保持兼容）
  - 虚函数指针占用一个指针size
  - 静态数据成员不参与
- 内存对齐与填充——
  - 对象内存对齐是为了优化CPU存储数据效率、避免数据截断
  - 按对齐系数（4字节、8字节）整倍数
  - 可使用#pragma pack(4)控制
  - 简单优化：长字段放前，短字段置后（聚集）
- 对象有多大？sizeof

特殊成员函数与三法则（Rule of Three）

- 四大特殊成员函数
  - 默认构造函数（无参），如果不定义任何拷贝构造，编译器自动生成
  - 析构函数/ 拷贝构造函数 / 赋值操作符，如果不定义，编译器自动生成
  - 使用 default 让编译器自动生成。
  - 使用 delete 让编译器不要自动生成。
- 三法则：析构函数、拷贝构造函数、赋值操作符 三者自定义其一，则需要同时定义另外两个（编译器自动生成的一般语义错误）。
- 编译器自动生成的拷贝/赋值是按字节拷贝，如不正确，则需要自定义拷贝/赋值/析构行为：
  - 赋值操作符中的 Copy & Swap 惯用法
  - 注意赋值操作中避免“自我赋值”。
- 需要自定义三大函数的类，通常包含指针指向的动态数据成员。

清楚对象的构造/析构点

- 构造器什么时候被调用？
  - 当对象（包括嵌套成员）被定义时（堆栈、堆或静态）。
  - 当对象的数组被定义时（堆栈、堆或静态）。
  - 当函数参数以值传递时
  - 当函数返回一个对象时
  - 甚至适用于编译器生成的临时对象。
- 析构器什么时候被调用？
  - 当命名的堆栈对象、数组或参数超出范围时（包括嵌套对象成员）
  - 当堆对象或数组被删除时
  - 对于静态对象，在程序结束时
  - 对于临时对象，在创建它们的 "完整表达式 "结束时调用

性能指南——类型与成员（1）

- 如果函数非常小，并时间敏感，将其声明为 inline
- 如果函数可能在编译期进行求值，就将其声明为 constexpr
- 类定义中禁止不期望的复制
- 使用成员初始化式来对类内数据成员进行默认值初始化
- 如果定义或者 =delete 了任何复制、移动或析构函数，请定义或者 =delete 它们全部（五法则）
- 将单参数的构造函数声明为 explicit，复制和移动构造函数则不
- 采用 union 用以节省内存

性能指南——类型与成员（2）

- 通过常量引用传递只读参数，而不是通过值。
- 尽可能地推迟对象的定义：晚加载，早释放
- 优先在构造函数中进行初始化而不是赋值。
- 考虑重载以避免隐式类型转换。
- 理解返回值优化（RVO）

常用类型——数组最佳实践

- 尽量避免使用C风格数组，有很多安全隐患
  - 本质是指针指向的一块连续内存，引用语义
  - 不带长度信息，易错点：拷贝、传参、返回值
- 不要使用指针传递数组，传递指针仅代表单个对象
- 不要使用C风格数组承载多态对象（基类、子类）

- 使用抽象管理内存
  - 使用vector<T>实现变长数组，替代堆上的C风格数组
  - 使用array<T>实现定长数组，替代栈上的C风格数组

常用类型——字符串最佳实践

- 尽量避免使用C风格字符串(“”默认字面常量）
  - 零结尾的字符数组，与字符数组有区别
  - 拥有一切C风格数组的安全隐患
- 使用string替代C风格字符串(“”字面常量以s结尾)
  - 能够正确分配资源，处理所有权、拷贝、扩容等操作
  - 但谨慎处理C风格字符串API和string的交互
- string内部实现了短字符串优化技术，拥有极高性能
  - 短字符串（<14字符）默认存在栈中，长字符串将分配于堆上。
- string_view 字符串的只读视图，表示为(指针,长度)
  - 不拥有,，不拷贝字符串，是字符串只读操作的性能之选

戒除C语言的“不良习惯”

- 辨析使用场景和对象所有权，谨慎使用裸指针
- 使用C++类型转换替换C风格的强制转换
- 使用模板和编译时计算等替换C风格的宏机制
- 尽量避免全局数据，谨慎使用全局函数
- 严格避免 malloc() 和 free()
- 当一定要用 C 时，应使用 C 和 C++ 的公共子集，并将 C 代码以 C++ 来编译

其他类型——枚举类

- enum class 是一种限制了作用域的强类型枚举
  - 强类型，不能和整数类型进行隐式转换。
  - 可以显式指定枚举类的基础类型（存储类型），默认int
  - 可以使用整数常量初始化枚举值
  - 和整数之间转型要使用显式转型（static_cast）
- 普通enum类型继承自C语言
  - 弱类型，和基础类型存在隐式转换
  - 作用域位于enum本身所在作用域（名字空间污染）
- 建议使用enum class替换不安全的enum

其他类型——联合与位域

- union 联合：所有成员分配在同一起始地址
  - size为最大成员的大小。同一时刻只能保存一个值。
  - 不要使用联合做类型转换（危险、且无法移植）
  - union 要谨慎使用，谨慎评估是否真的会带来性能提高。
  - union成员状态很容易破坏，有时使用继承来替代union是更好的选择。
- bit-field 位域：指定结构成员所占用的具体bit位数
  - 成员必须是整数或者枚举类型。
  - 无法获取成员的地址。
  - 成员从上到下，地址从低到高按位依次排列。
  - 谨慎评估性能，节省了数据存储的空间，但管理和操作代码复杂化，代码段更长，常常得不偿失。访问char或者int也更快（内存对齐）。