Morden cpp 30 lecture — heap and stack

Monday, December 14, 2020 1:42 PM

Why C++:

C++是一门多范式的通用编程语言.
- 多范式：面向过程，面向对象，泛型编程，函数式编程.

用到C++的例子：
- Google Chrome
- StarCraft
- 火星车的自动驾驶系统.

C++的核心竞争力：
- 抽象能力
- 性能
- 功耗.

极客时间：现代C++实战

基本概念

堆 heap：动态分配内存的区域，被分配后要手动释放，否则造成内存泄漏
- new和delete操作的区域是free store
- malloc和free操作的区域是heap

栈 stack：函数调用过程中产生的本地变量和调用数据的区域
- LIFO

RAII：resource acquisition is initialization，是C++特有的内存管理方式.

堆：
- Eg:

```
  auto ptr = new std::vector<int>();
```

- 让内存管理器分配一个某个大小的内存块
- 让内存管理器释放一个之前分配的内存块
- 让内存管理器进行垃圾收集，寻找不再使用的内存块并予以释放.

```
void foo()
{
  bar* ptr = new bar();
   ...
   delete Ptr;
}
```

- 中间省略的代码也许会抛出异常，导致最后的delete ptr不执行.
- 不符合C++惯用法.在C++中，一般使用栈分配内存，不使用堆.

```
bar* make_bar(...)
{
  ...
  try {
    bar* ptr = new bar();
    ...
  }
```

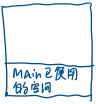
```
  catch(...){
    delete ptr;
    throw;
  }

  return ptr;
}

Void foo()
{ ...
  bar* ptr = make_bar(...);

  delete ptr;
}
```

更常见的是,分配和释放不在一个函数里,更容易遗忘delete.

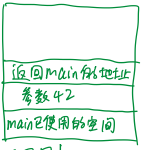
栈

- Eg.

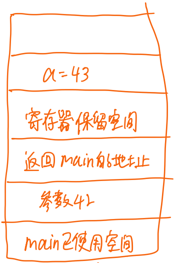
```
Void foo(int n)
{
  ...
}

Void bar(int n)
{
  int a = n+1;
  foo(a);
}

int main()
{
  ...
  bar(42);
  ...
}
```



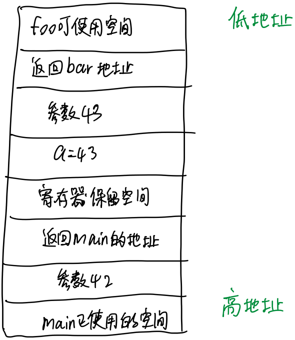
执行bar之前



调用bar



执行foo之前



调用 foo

- 本地变量所需内存就在栈上,跟函数执行时所需的其他数据在一起.
- 由于是后进先出的执行过程,不可能出现内存碎片.
- 某个函数占用的栈空间,叫作栈帧stack frame.

前面调用的是简单类型的本地变量,C++里称为PoD (plain old data),
对于有构造和析构函数的非POD类型,编译器会自动调用析构函数,
包括在函数执行发生异常的情况,
- 在异常时对析构函数的调用,有个专门术语:栈展开Stack unwinding

```
#include <stdio.h>

class Obj{
```

```
public:
    Obj() { puts("Obj()"); }
   ~Obj() { puts("~Obj()"); }
};

void foo(int n)
{
  Obj obj;
  if(n == 42)
    throw "Life, the universe and everything.";
}

int main()
{
  try {
     foo(41);
     foo(42);
  }
  catch(const char* s){
     puts(s);
  }
}
```

执行结果:

```
Obj()
~Obj()
Obj()
~Obj()
Life, the universe and everything.
```

不管是否发生了异常, obj 的析构函数都会执行.

- 在C++中, 所有变量缺省都是值语义.
- 使用*和&, 才会引用一个堆上的对象.
- 对于智能指针ptr, 用ptr→call() 和 ptr.get() 语法上都对.

RAII

在很多情况下, 对象不应存在栈上:

- 对象很大
- 对象的大小在编译时不确定
- 对象是函数的返回值, 但不使用值返回.
- Eg. 工厂方法中, 返回值类型是基类.

```
enum class shape_type{

   Circle,
   triangle,
   rectangle,
   ...
};
```

```
Class shape {...};
class circle: public shape {...};
Class triangle : public shape {...};
class rectangle : public shape {...};

shape* create_shape (shape_type type)
{
  ...
  switch(type) {
  case shape_type::circle:
      return new circle(...);
  case shape_type::triangle:
      return new triangle(...);
  case shape_type::rectangle:
      return new rectangle(...);
  ...
  }
}
```

create_shape 方法会返回一个shape对象，对象的实际类型是某个shape的子类，eg 圆，三角，矩形，这时函数的返回值只能是智能指针或其变体形式.
如果返回类型是shape，实际返回 Circle，编译器不报错，但结果是错的．叫对象切片 object slicing

要确保使用 create-shape 的返回值时不会发生内存泄漏，需要把返回值放在一个本地变量里并确保其析构函数会删除该对象

```
class shape_wrapper{

public:
   explicit shape_wrapper(
      shape* ptr = nullptr)
      : ptr_(ptr) {}
  ~shape_wrapper()
  {
    delete ptr_;
  }
  shape* get() const {return ptr_;}

private:
  shape* ptr_;
};

void foo()
{ ...
   shape_wrapper ptr_wrapper(
      create_shape(...));
  ...
}
```

关于class constructor中的explicit关键字:

```
#include <iostream>

using namespace std;

class Complex:
{
  private:
    double real;
    double imag;

 public:
    //Default constructor
    Complex(double r=0.0, double i = 0.0):
       real(r),
       imag(i)
       {}

    // A method to compare two Complex numbers
    bool operator==(Complex rhs){
       return (real == rhs.real && imag== rhs.imag)? true: false;
    }
};

int main()
{
   // a complex object
   Complex com1(3.0, 0.0)

   if(com1 == 3.0)
      cout << "Same";
   else
      cout << "not same";
   return 0;
}
```

Output: same.

In C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor, because such a constructor allows conversion of the single argument to the class being constructed.

We can avoid such implicit conversions as these may lead to unexpected results.

```
#include <iostream>

using namespace std;

class Complex
{
private:
    double read;
    double imag;

public:
```

```
    //Default constructor
    explicit Complex(double r = 0.0, double i = 0.0) :
        real(r),
        imag(i)
    {}

    //A method to compare two complex numbers
    bool operator == (Complex rhs) {
        return (real == rhs.real && imag == rhs.imag) ? true : false;
    }
};

int main()
{
    //a complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "not Same";

    return 0;
}
```

Output:

Compiler Error:

No match for 'operator==' in 'com1 == 3.0 e to'.

We can still typecast the double value to Complex, but now we have to explicitly double cast it. i.e.

```
if(com1 == (Complex) 3.0)
    cout << "Same";
else
    cout << "not Same";

return 0;
```

在new一个对象和delete对象时,

```
// new Circle(...)
{
    void* temp = operator new(sizeof(Circle));
    try {
        Circle* ptr = static_cast<Circle*>(temp);
        ptr->Circle(...);
        return ptr;
    }
    catch(...) {
        operator delete(ptr);
```

```
    throw;
  }
}
//delete:
if (ptr != nullptr) {
   ptr -> ~shape();
   operator delete(ptr);
}
```

- new的时候先分配内存，失败时整个操作失败并向外抛出异常，e.g. bad_alloc.
- 然后new在结果指针上构造对象，成功则new操作完成，否则释放内存并抛出异常.
- delete则断判指针是否为空，在指针不为空时调用析构函数并释放内存.

在析构函数里做必要的清理工作，是RAII的基本用法，包括
- 释放内存
- 关闭文件（fstream析构时）
- 释放同步锁
- 释放其它重要系统资源.
- Eg.

```
  Std::mutex mtx;

  Void some_func()
 {
   std::lock_guard<Std::mutex> guard(mtx);

     ...
 }
```

而不是

```
Std::mutex  mtx;

Void some_func()
{
  mtx.lock();
  ...
  mtx.unlock();
}
```

如果发生异常或提前返回，下面这句不会自动执行.