18 Memory model and Dynamic Memory
- 18.1 Global(Static), Local(automatic) and dynamic variables
  - 18.1.1 Global(static) variables
  - 18.1.2 Local (Automatic) variables
  - 18.1.3 Dynamic variables
  - 18.1.4 使用 dynamic variables 的原因
- 18.2 Dynamic Memory 的使用
  - 18.2.1 `new`
  - 18.2.2 `delete`
    - 18.2.2.1 `assert(nullptr)`
  - 18.2.3 Pitfall: double `delete` 是 undefined behavior
  - 18.2.4 Pitfall: memory leak!
- 18.3 Program Segments
  - 18.3.1 The little void
  - 18.3.2 Text segment
  - 18.3.3 Global segment
  - 18.3.4 Heap segment
  - 18.3.5 The BIG VOID(未使用区域) 以及 Stack
- 18.4 Dynamic arrays
  - 18.4.1 `delete[]`

18 Memory model and Dynamic Memory

18.1 Global(Static), Local(automatic) and dynamic variables

回顾：

1. 一个 object 是 memory 中的 a piece of data.
2. 一个 object lives at an address in memory
3. 一个 object 只能在其 lifetime 被使用，lifetime 取决于 storage duration.
4. c++ 中有三种 storage durations:
   - Global / static
   - Local / automatic
   - Dynamic

18.1.1 Global(static) variables

有三种：

1. 第一种: 在 function/class definition 外的 globle variables. 在 program 运行前就被创造，一直生存到 program 结束。

```
const int SIZE=10;
int main() {
  //...
}
```

2. 第二种: `static` within a class definition.

```
class Set {
private:
  static const int CAPACITY = 100;
  //...
};
```

3. `static` within a function

```
int main() {
  static const int SIZE = 10;
  //...
}
```

18.1.2 Local (Automatic) variables

A local variable lives inside a block `{ }`.

在 declaration 被 run 的时候被 initialized. 在 block finish 的时候 die.

18.1.3 Dynamic variables

Lifetime 在 runtime 被决定.

Programmer 决定它被 created 以及 destroyed 的时间.

18.1.4 使用 dynamic variables 的原因

```
int * pointerToThree() {
  int x = 3;
  return &x;
}

int main() {
  int *z = pointerToThree();
  cout << *z;
}
```

这段代码 error. 因为 `pointerToThree()` 创造的 `x` 及其地址作为的指针变量是一个 local variable，在 `pointerToThree()` 运行结束的时候就无了。因而这片地址上不再是一个 `int` 了，所以 type error.

因而我们需要 dynamic variable. 我们可以自主控制其 lifetime，这样 pointers to the same object can exist in different scopes.

除了这个好处外，heap 的容量大小也比 stack 要远大。

18.2 Dynamic Memory 的使用

18.2.1 `new`

`new` 在 dunamic memory 中**创造一个 object，并 return 指向这个 object 的一个 pointer.**

```
int * pointerToThree() {
  int *p = new int;
  *p = 3;
  return p;
}
```

```
// 或者
int * pointerToThree() {
  int *p = new int(3);
  return p;
}
```

在这里，**`p` 是 `pointerToThree` 的 stack frame 中的一个 local variable！**它的类型是 `int *`. 一旦 `pointerToThree` 运行结束，它就 die 了.

而值为 `int(3)` 的变量则是在 heap 的 dynamic memory 中的一个 `int` 型 variable。我们将用 `int(3)` 来称呼它。我们通过使用 `new` 把 heap 的一段 memory 给 allocate 给了它。**在 `pointerToThree` 运行结束后，它仍然存在！**

`pointerToThree` 运行结束之后，`p` die 了，`p` 的 memory 被自动 release 了；但是 `p` 的值，也就是 `int(3)` 的地址，被 return 了。

只要我们不手动 `delete`，`int(3)` 将始终在 dynamic memory 直到 program 结束！

现在我们调用 `pointerToThree`，将它返回值分配给 `main` 中的变量。

```
int main() {
  int *z = pointerToThree();
  cout << *z;
  //...
}
```

于是 `p` 的值在 `p` 死后被 return 给了 `main` 中的 `int *` 变量 `z`.

18.2.2 `delete`

`delete` 可以 destroy dynamic variables.

```
int main() {
  int *z = pointerToThree();
  cout << *z;
  delete z;
}
```

需要注意的是：我们写的是 `delete z`，`z` 是一个 `main` 的 stackframe 中的一个 local variable，但其实它做的是 **destroy `z` 所指向的 dynamic memory 中的 dynamic object**，也就是值为 `int(3)` 的那个变量！

在我们的 delete 之后：

1. **`z` 仍然指向 heap 中的同一位置，`z` 的值并不会改变！！** 但此时我们已经不能 dereference 它了，这会是一个 undefined behavior.
2. 我们可以重新给 `z` 赋值

我们注意到有一个比较抽象的事情：

我们发现在 `delete` 后 dereference `z` 来获取已经释放的 dynamic memory 上的东西仍然是可以的，**只是一个 undefined behavior 而不是一个 error.**

理论上，在 `delete` 以后，该 object 所占用的 memory 被归还给 operating system 了，用于未来的 allocation。从逻辑上该 object 不再存在.

然而，在物理层面上，该 memory 在被 `delete` 之后与立即重用之前，其内容可能仍然保留未更改的状态，就是说这个位置的值可能仍然是一个 `int(3)`. 然而此时 dereferene 是 undefined behavior，是不安全的。

18.2.2.1 `assert(nullptr)`

有一个好的做法给指向已经删除的 dynamic object 的 ptr 变量分配 `nullptr`，这样首先这个 ptr 就不再是一个 dangling pointer 了，其次我们可以在合适的地方 `assert` 它来 identify problems.

注意：`assert(nullptr)` 的结果是 segfault，因而它有助于勘探问题。

```
int main() {
  int *p = new int(100);
  delete p;
  p = nullptr;
  assert(p); //SegError
}
```

18.2.3 Pitfall: double `delete` 是 undefined behavior

```
int main() {
  int *p1 = new int(3);
  int *p2 = p1; // p2 现在也指向该 dynamic object
  delete p1;  // delete 了这个 dynamic object
  delete p2; // double delete, Undefined behavior!
}
```

这里 `p1` 和 `p2` 指向的是同一个 dynamic object，因而只需要也只至多应该 `delete` 一次。如果我们忘记了，前后 `delete` 了 `p1` 又 `delete` 了 `p2`，那么就会导致 undefined behavior，并且很可能 runtime error.

18.2.4 Pitfall: memory leak!

```
int main() {
  int *p1 = new int(1);
  int *p2 = new int(2);
  p1 = p2; // 现在没有东西指向值为 int(1) 的那个 dynamic object 了
  delete p1;
}
// how much memory is leaked? sizeof(int) == 4B.
```

Dynamic variables 会一直存在直到被 `delete`，即便已经没有东西指向它了。

这也意味着：**我们如果不小心把指向一个 dynamic object 的 ptr 变量的值改掉，让它不再指向这个 dynamic object，那么我们就再也无法 `delete` 它了。**这就叫做一个 **memory leak.**

不仅如此，有很多个渠道能够导致这种事情。比如指向 dynamic object 的 ptr 变量出了自己的 scope 死了，但是并没有把值传递出去。

```
void f() {
  int *p = new int(1);
  cout << *p;
}

int main() {
  f();  // memory leak
}
```

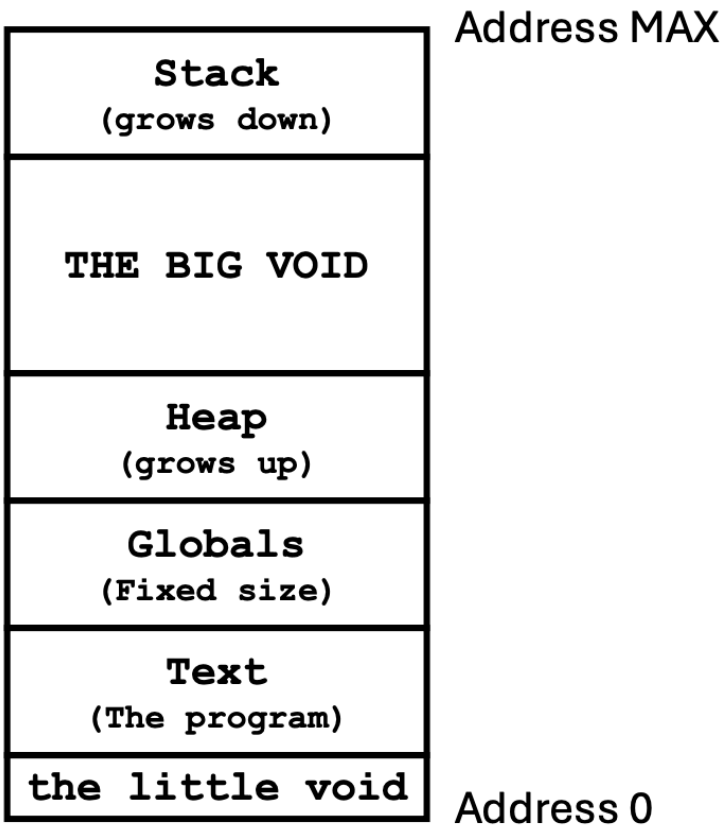
18.3 Program Segments

一个 running program 有一个 "address space", 就是一系列 memory locations that are accessible to it.

每个 address space 都 private 于一个 running program，没有其他的 running program 能够 access 或者 modify 它！

除了 heap segment 之外，一个 running program 还有其他的 segments.

Operating system manages **virtual memory**，使得**每个 program 都能够装作它 occupies 整个 address space**，从 address 0 到 address max.



18.3.1 The little void

大多数 operating systems 都 reserve 了从 0 开始的前几千个 addresses，这部分构成了 **the little void**.

(Ps: address 0 也就是所有 nullptr 存储的地址)

18.3.2 Text segment

这个 segment 包含的是 the machine code (也就是 binary)，包含了整个 compiled program 的 binary code。它的 address 所在位置也很低，仅在 little void 之后。

18.3.3 Global segment

space for the global variables 以及 static variables.

18.3.4 Heap segment

`new` 和 `delete` 是 c++ 中我们 interact with heap segment 的方法，而 C 中我们也有类似的方法，`malloc()` 和 `free()` 就是 C 的 `new` 和 `delete`.

heap segment 中，**每个 heap object 都有自己的 lifetime，与它周围的其他 objects 无关！** 不像 stack frame 中，object 的 lifetime 在其所在的 `stack frame` 被建立时开始，被 destroy 时结束。

18.3.5 The BIG VOID(未使用区域) 以及 Stack

The BIG VOID 是在 heap segment 和 stack segment 之间的区域， for the stack and heap to grow. Stack 就是我们熟悉的栈。Stack 在所有 segments 的最顶上。

18.4 Dynamic arrays

我们平时用的 **statically allocated array 的 size 必须是在 compile time 就确定的**，这是 C++ standard. 但是 **dynamically allocated array 的大小可以在 runtime 被确定。**

```
int howMany;
cin >> howMany;
int *arrPtr = new int[howMany];
```

`new` 创建的 dynamic array 是一个指向它首元素的 pointer.

但是我们仍然可以使用 `[]` operator 来获取元素。

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
  for (int i = 0; i < howMany; ++i)
    arrPtr[i] = 42;
}
```

18.4.1 `delete[]`

有一个问题是：runtime environment 是无法 distinguish 在 heap 上指向单个元素和指向 dynamic array 的首元素的指针的。

因而我们需要使用一个特别的 `delete[]` syntax for arrays.

```
int main() {
  int *ptr1 = new int(42);
  int *ptr2 = new int[howMany];
  //...
  delete ptr1;
  delete[] ptr2;
}
```