New and delete

C'de memory allocation için malloc() ve free() kullanıyorduk, malloc ve free C++ da da kullanılabilir, ancak bunu yapmak iyi bir fikir değildir. Çünkü bir obje için malloc ile yer ayırdığınızda Constructor çağrılmayacaktır. Aynı şekilde, free kullandığımızda Destructor çağrılmaz. Bu yüzden, C++ objeleri için allocation yaparken bunun yerine new ve delete kullanacağız.

```
class Student
{
  private:
    std::string _login;
  public:
    Student(std::string login) : _login(login)
    {
      std::cout << "Student" << this->_login << "is born" << std::endl;
    }
    ~Student()
    {
      std::cout << "Student" << this->_login << "died" << std::endl;
    }
```

```
int main()
{
  Student bob = Student("bfubar");  → stack allocated
  Student* jim = new Student("jfubar")  → Heap allocated
                                          new kullanarak

  delete jim  → new kullanarak allocate ettiğimiz objeyi
                delete kullanarak de-allocate etmeliyiz.
                Bu şekilde allocate edilen objeler scope
                bittiğinde yok edilmez, bu yüzden delete
                kullanmazsanız leak ile karşılaşırsınız.

  return(0);
}  → Scope sonu, bob otomatik olarak yok edilir.
```

```
class Student
{
  private:
    std::string _login;
  public:
    Student() : _login("Idefault")
    {
      ...
    }
    ~Student()
    {
      ...
    }
```

New ve delete kullanarak array allocate edebiliriz.

```
int main()
{
  Student* students = new Students[62];
    ...
  delete students;
}
```

new, içinde 62 tane Student örneği olan bir array allocate eder. new ile allocate ettiğimiz arraylerde parametre kullanamadığımızı dikkat edin. Bu şekilde oluşturulan objeleri initialize etmek için çözümler bulmamız gerekecek. Bu örnekte sadece tüm loginlere Constructorda default bir değer verdik.

References

C'de, scope dışında görülebilen değişkenleri heap'te allocate etmek için pointerları kullanıp, adresini kullanarak değişkeni manipüle edebiliyorduk. Her yerde * ve & kullanıyorduk ve bu pek pratik değildi.

C++'da benzer işi gören bir konsept var: Referanslar. Bir referans, kabaca var olan bir değişkenin alias'ı (diğer ismi) olarak tanımlanabilir.

```
int main()
{

    int numberOfBalls = 42;

    int* ballsPtr = &numberOfBalls → pointer
    int& ballsRef = numberOfBalls → reference

    ↓

    Referansa, değişkenin adresini değil, kendisini atmamız gerekir.
    Artık ballsRef referansı numberOfBalls değişkenine işaret ediyor.
    Bir referans const (sabit) bir pointer gibidir, her zaman aynı
    değişkene işaret eder ve re-assignment yapılamaz.

    std::cout << numberOfBalls << " " << *ballsPtr << " " << ballsRef << std::endl;

    *ballsPtr = 21;
    std::cout << numberOfBalls << std::endl;

    ballsRef = 84
    std::cout << numberOfBalls << std::endl;

    return(0);

}
```

Output:

```
42  42  42
21
84
```

```
int & ballsRef2;
```

Bu şekilde initialize etmeden referans declare edemezsiniz, çünkü referanslar bir şeye işaret etmek zorundadır. Referansı sonradan değiştirmek de imkansız olduğu için declare ettiğiniz satırda initialize etmelisiniz.

Ayrıca, sonradan de-allocate edilme olasılığı olduğu için, referanslar memory alanlarına işaret etmemelidir.

Passing parameters by reference

```
void byPtr(std::string* str)
{
    *str += " and ponies";
}

void byConstPtr(std::string const *str)
{
    std::cout << *str << std::endl;
}

void byRef(std::string& str)
{
    str += " and ponies";
}

void byConstRef(std::string const& str)
{
    std::cout << str << std::endl;
}
```

```
int main()
{
    std::string str = "I like butterflies";

    std::cout << str << std::endl;
    byPtr(&str);
    byConstPtr(&str);

    str = "I like otters";

    std::cout << str << std::endl;
    byRef(str);
    byConstRef(str);

    return(0);
}
```

Output:

```
i like butterflies
i like butterflies and ponies
i like otters
i like otters and ponies
```

```
class Student
{
  private:
    std::string _login;
  public:
    Student(std::string const& login) : _login(login)
    {
    }
    std::string& getLoginRef()
    {
      return this->_login;
    }
    std::string const& getLoginRefConst() const
    {
      return this->_login;
    }
    std::string* getLoginPtr()
    {
      return &(this->_login);
    }
};
```

Referansları yukarıdaki gibi fonksiyon dönüş değeri olarak kullanabiliriz. Bu şekilde, dönen referans aracılığıyla memberları manipüle edebiliriz. Ancak const referanslar obje üzerinde değişiklik yapmazlar.

```
int main()
{
    Student bob = Student("bfubar");
    Student const jim = Student("jfubar");

    std::cout << bob.getLoginRefConst() << " ";
    std::cout << jim.getLoginRefConst() << std::endl;

    std::cout << *(bob.getLoginPtrConst()) << " ";
    std::cout << *(jim.getLoginPtrConst()) << std::endl;

    bob.getLoginRef() = "bobfubar";

    std::cout << bob.getLoginRefConst() << std::endl;

    *(bob.getLoginPtr()) = "bobbyfubar";

    std::cout << bob.getLoginRefConst() << std::endl;

    return 0;
}
```

→ fonksiyondan dönen referansı kullanarak bob'un _login memberını değiştirebildik.

Output:

```
bfubar jfubar
bfubar jfubar
bobfubar
bobbyfubar
```

! Referanslar kullanışlıdır ve pointerlara göre avantajları vardır ancak pointerların yerini tutmazlar.

Referansları declare ettiğimiz gibi initialize etmek zorundayız ve bir referansın işaret ettiği değişkeni değiştiremeyiz. Bu yüzden bazı durumlarda pointer kullanmak daha mantıklıdır.

Örneğin diyelim ki bir RPG karakteri classınız var. Bu karakterin bir silahı olabilir ya da olmayabilir. Karakter silahını düşürebilir, yeni bir tane alabilir vs. Bu durumda bir referansa değil bir pointera ihtiyacımız olur.

Bir şeyin ilk başta uninitialized olması veya sonradan değişmesi gerekiyorsa pointer, her zaman var olması ve adresinin değişmemesi gerekiyorsa referans kullanmak daha mantıklıdır diyebiliriz.

Filestream

Daha önce stdout ve stdin için kullanılan streamleri görmüştük. C++da dosyalarla etkileşime girmek için kullanılan streamler de vardır.

```
#include <fstream>

int main()
{
    std::ifstream ifs("numbers");  → open input filestream for filename "numbers"
    unsigned int dst;
    unsigned int dst2;

    ifs >> dst >> dst2;

    std::cout << dst << " " << dst2 << std::endl;

    ifs.close(); → close the filestream after done with it

    //----

    std::ofstream ofs("test.out"); → output filestream

    ofs << "i like ponies a whole damn lot" << std::endl;

    ofs.close(); → close
```

"numbers" file content:

```
8 12
```

Output:

```
8 12
```

test.out:

```
i like ponies a whole damn lot
```

Filestreamleri kullanmanın bir çok yolu var araştırmak size kalmış.

cplusplus.com / doc/tutorial / files