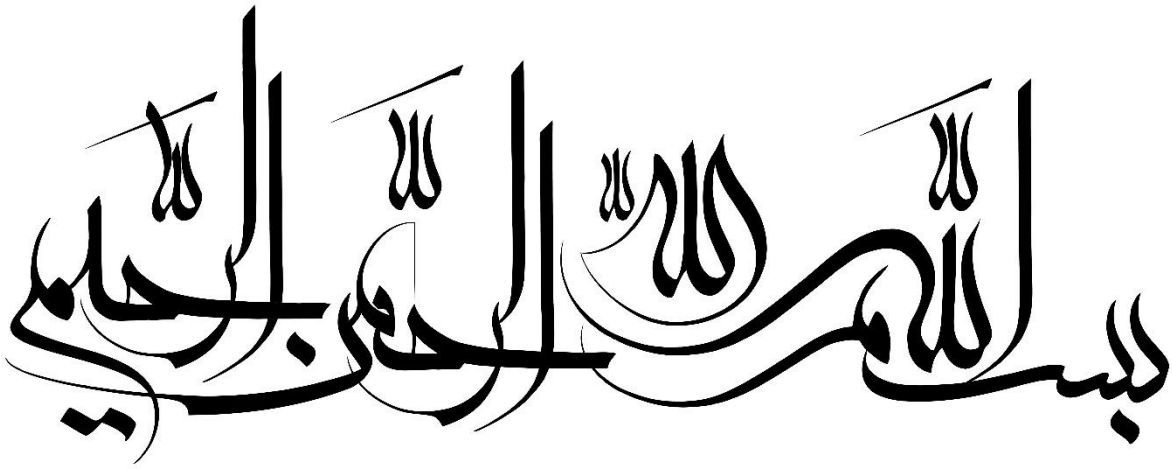

عنوان : constructer و deconstructer

& Gc Collect

استاد: جناب آقای دکتر میثاق یاریان

دانشجو: سیدمحمد موسوی مطلق

درس: برنامه نویسی سمت سرور

پاییز 1402

سازنده کلاس

سازنده کلاس (Constructor) یک تابع عضو ویژه از کلاس می‌باشد که هنگام ایجاد یک شی جدید از آن کلاس اجرا می‌شود.

سازنده دقیقا هم‌نام کلاس می‌باشد و هیچ نوع داده‌ای) حتی (void را برنمی‌گرداند. سازنده‌ها در مقداردهی اولیه به متغیرهای عضو بسیار مفید هستند.

مثال زیر مفهوم سازنده را توضیح می‌دهد.

```
#include <iostream>

using namespace std;

class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line();  // This is the constructor
  private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void) {
  cout << "Object is being created" << endl;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

// Main function for the program
int main() {
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
```

```
  return 0;
}
```

.وقتی که کد فوق کامپایل و اجرا شود، خروجی زیر حاصل می‌شود

```
Object is being created

Length of line : 6
```

سازنده‌ی دارای پارامتر(Parameterized Constructor)

سازنده‌ی پیش‌فرض هیچ پارامتری ندارد، اما اگر نیاز باشد، سازنده‌ها می‌توانند پارامتر هم داشته باشند. با این روش می‌توان هنگام ایجاد یک شی جدید، مقادیر اولیه‌ای به آن اختصاص داد. مثال زیر را ملاحظه کنید.

```
#include <iostream>

using namespace std;
class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line(double len);  // This is the constructor

  private:
    double length;
};

// Member functions definitions including constructor
Line::Line( double len) {
  cout << "Object is being created, length = " << len << endl;
  length = len;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

// Main function for the program
```

```
int main() {
  Line line(10.0);

  // get initially set length.
  cout << "Length of line : " << line.getLength() <<endl;

  // set line length again
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

با اجرای این کد، خروجی زیر حاصل می‌شود.

```
Object is being created, length = 10

Length of line : 10

Length of line : 6
```

استفاده از لیست آغازین (Initialization List) برای مقداردهی اولیه فیلدها

برای سازنده‌های دارای پارامتر، از ساختار زیر برای مقداردهی اولیه فلیدها استفاده می‌شود.

```
Line::Line( double len): length(len) {

  cout << "Object is being created, length = " << len << endl;

}
```

ساختار بالا مشابه ساختار زیر می‌باشد.

```
Line::Line( double len) {

  cout << "Object is being created, length = " << len << endl;

  length = len;

}
```

اگر در کلاس C فیلدهای X، Y، Z و ... وجود داشته باشد، برای مقداردهی اولیه آنها می‌توان از دستور زیر استفاده کرد، ملاحظه کنید که نام فیلدها با کاما از هم جدا شده اند.

```
C::C( double a, double b, double c): X(a), Y(b), Z(c) {

  ....

}
```

تخریب‌کننده‌های کلاس

تخریب‌کننده (Destructor) یک تابع عضو ویژه از کلاس است و هنگامی اجرا می‌شود که شی آن کلاس از دامنه خارج شده و یا دستور delete به اشاره‌گر آن اعمال گردد.

یک تخریب‌کننده دقیقاً هم نام با کلاس است با این تفاوت که یک (~) پیش از نام آن قرار می‌گیرد. این تابع هیچ مقداری را برنمی‌گرداند و هیچ پارامتری را نیز دریافت نمی‌کند. تخریب کننده‌ها در آزادسازی منابع قبل از خروج از برنامه، یعنی بستن فایل‌ها، آزادسازی حافظه و... بسیار مفید هستند.

مثال زیر مفهوم تخریب کننده را توضیح می‌دهد.

```
#include <iostream>

using namespace std;
class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line();   // This is the constructor declaration
    ~Line();  // This is the destructor: declaration

  private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void) {
  cout << "Object is being created" << endl;
```

```
}
Line::~Line(void) {
  cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

// Main function for the program
int main() {
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

با اجرای این کد خروجی زیر حاصل می‌شود.

```
Object is being created

Length of line : 6

Object is being deleted
```

Garbage Collector چیست و چه انواعی دارد؟

در سال های ابتدایی برنامه نویسی معمولا دانشجو ها با برنامه های کوچک سر و کار دارند. در سال های ابتدایی کار هم معمولا بخش های کوچکی از یک برنامه را می نویسند و یا با زبان های سطح بالا کار می کنند که مدیریت خودکار حافظه دارند. در زبان هایی مثل C/C++ Assembly و ... شما باید خودتان حافظه را مدیریت کنید و اگر به حافظه پویا dynamic memory نیاز داشته باشید باید خودتان آن را از سیستم عامل بگیرید. در زبان هایی که مدیریت حافظه خودکار دارند مثل Java, C#, Python, Go و ... مدیریت حافظه برای شما به طور خودکار صورت می گیرد.

بخشی از runtimeی که برنامه شما را اجرا می کند، مثلا virtual machine مربوط به Java و یا CLR (.NET) به نام Garbage Collector یا همان GC برای شما حافظه را مدیریت می کند. این کار با دو الگوریتم کلی در زبان های تجاری و غیر دانشگاهی اتفاق می افتد. یک نوع Mark and sweep GC (tracing) و نوع دیگر Automatic reference counting GC . بیشتر زبان های معروف از نوع اول استفاده می کنند و زبان هایی مثل Objective-C و Erlang از Automatic reference counting استفاده می کنند. در زبان هایی مثل سی پلاس پلاس و برخی کتابخانه ها و framework های سیستم عامل ها (مثل COM) از reference counting برای مدیریت برخی منابع استفاده می شود.

در روش reference counting شما برای هر assignment و خارج شدن متغیر ازscope یا ست کردن متغیر به null یکی به تعداد reference های یک Object اضافه و یا کم می شود. هر گاه تعداد reference های یک Object صفر شود می توانید آن را از حافظه پاک کنید. این روش باعث می شود هر assignment و خروج متغیر از scope کندتر از یک زبان mark and sweep GC اتفاق بیفتد ولی به طور کلی هر کدام از این دو روش می توانند در شرایط مختلف سریعتر از روش دیگر باشد. در این روش یک فیلد در حافظه خود object تعداد reference ها را می شمارد و پس از صفر شدن destructor را صدا زده و سپس object حافظه خود را به سیستم پس می دهد. در این روش اگر reference حلقه ای بین object ها وجود داشته باشد دیگر هیچ کدام از آن ها از حافظه خارج نمی شوند. به همین دلیل نوع دیگری از reference به نام weak ereference وجود دارد که باعث زیاد شدن تعداد reference های object نمی شود ولی object از وجود آن با خبر است و در پایان عمرش آن را به null ست می کند. این روش به طور کلی کمی سرعت اجرای برنامه را پایین می آورد و در عوض میزان کارایی برنامه کاملا deterministic بوده و هرگز وسط اجرا pause اتفاق نمی افتد و همه thread ها برنامه را اجرا می کنند.

در روش Mark And Sweep یا Tracing محیط اجرای برنامه یا همان runtime هنگام تخصیص حافظه به تمام متغیر های static و متغیر های عمومی و متغیر های قابل دسترسی در گراف برنامه از ریشه نگاه می کند و چک می کند کدام object از لیست object های allocate شده دیگر قابل دسترسی نیست و آن را پاک می کند. این روش می تواند باعث توقف اجرای برنامه

یا حد اقل مشغول شدن بعضی از thread ها شود. معمولا این کار در زبان های پیشرفته مثل جاوا و سی شارپ به شکل generational صورت می گیرد که سرعت کار را بالا می برد.

در یک generational GC آبجکت ها ابتدا در gen 0 قرار می گیردند و هنگام اسکن کردن برای یافتن حافظه اضافه GC شما ابتدا gen 0 را بررسی می کند و اگر حافظه لازم را کسب کرد دیگر به اسکن لیست های بزرگتر gen 1 و gen 2 و ... نمی رود که احتمالا object های قدیمی در آن ها هستند که فعلا هم مورد استفاده و لازم هستند. اگر یک object یک بار در gen 0 زنده ماند به gen 1 منتقل می شود و اگر در یک اسکن از gen 1 زنده ماند به gen 2 می رود. این الگوریتم می تواند در محیط های مختلف به شکل های مختلف اجرا شود. در .NET شما دارای ۳ generation هستید که با اعداد ۰ و ۱و ۲ مشخص می شوند و object های بزرگتر از ۸۰k هم به Large Object Heap می روند و طقریبا مثل gen 2 با آن ها برخورد می شود.

برنامه نویسان Unity دقت کنند که این engine از آخرین نسخه GC در Mono استفاده نمی کند و همچنین GC مونو با دات نت فرق دارد و GC موجود در Unity اصلا generational نبوده و لیست همه Object ها را یک جا اسکن می کند و به این دلیل بسیار کندتر می باشد. دلیل این که Unity در حال معرفی Native Collection در C# Job system است خلاص شدن از شر GC است.