Урок 4

Ответы на контрольные вопросы(урок 4)

Какими способами можно задать значения для данных членов структуры

1) Присваивание значения для каждой из переменных

Тут все просто. Изначально поля структуры содержат в себе мусор или какие-то значения по умолчанию.
Наша задача - изменить эти значения. Например:
```
struct someStruct
{
   someStruct(int alpha,int betta)
   {
      this->alpha = alpha;
      this->betta = betta;
   }
   int alpha;
   int betta;
};
```
При создании данной струтуры, передаём необходимые значения в конструктор. В конструкторе значения полей изменяются на значения переданных переменных.

При использовании данного метода, мы затрачиваем много времени на бессмысленные операции:
- копирование переданных данных в функцию конструктора
- изменение значения полей конструктора

2) Использование списков инициализации

Список инициализации позволяет инициализировать поля структуры, при создании объекта структуры.

Например:
```
struct someStruct
{
   int alpha;
   int betta;
};
```
Используем список инициализации при создании нового объекта:
```
someStruct str = {5,2};
```
В результате данной операции значение переменной `alpha` станет равным пяти, `betta` - двум.

Этот способ оптимальнее предыдущего. Теперь мы не затрачиваем время на копирование переменных в дополнительную функцию.

Список инициализации в конструкторе структур

Из вышеописанного следует, что использование списков инициализации значительно ускоряет работу со структурами. Но очень часто синтаксис `someStruct obj = {data1,data2...}` не является удобным.

В таких случаях можем использовать список инициализации в конструкторе структуры. Синтаксис данной операции выглядит следующим образом:
```
```

```
constructor C(T param1,T2 param2): field1 {param1}, field2 {param2}
{

}
```

Какие же преимущества даёт данная конструкция? Сравним её с присваиванием значений полям в теле конструктора:
```
#include <iostream>

struct S1
{
   S1()
   {
      std::cout<<std::endl<<"default constructor"; //вызван конструктор по умолчанию
   }
   S1(int alpha): alpha { alpha }
   {
      std::cout<<std::endl<<"constructor 2"; //вызван конструктор 2 - не по умолчанию
   }

   int alpha;
};
struct S2
{
   S2(S1 someData)
   {
      this->someData = someData;
   }

   S1 someData;
};

int main()
{
   S1 a(20);
   S2 b(a);
   return 0;
}
```
При выполнении данного кода получаем сообщение:
> constructor2
> default constructor

Первое сообщение информирует нас о том, что у структуры `S1` был вызван конструктор 2. Это произошло при создании объекта `a`.

Второе сообщение означает, что в ходе программы был вызван конструктор по умолчанию структуры `S1`. Это произошло на моменте создания объекта `b` структуры `S2`.
Таким образом, перед присваиванием значения полю `someData` структуры `S2`, мы проинициализировали его используя конструктор по умолчанию.
Очевидно, что это действие является лишним. У структуры `S2` отсутствует коснтруктор по умолчанию, следовательно поле `someData` по любому будет проинициализировано во время воздания объекта структуры `S2`.

Теперь заменим конструктор `S2`. Будем использовать конструктор со списком инициализации:
```
S2(S1 someData): someData {someData}
{
```

```
}
```

Теперь, при выполнении кода, не заметим сообщения о вызове конструктора по умолчанию. То есть инициализация полей произошла до исполнения тела конструктора. При этом поля проинициализированы нужными для нас значениями.

Идиома RAII

RAII расшифровывается как Resource Acquisition is Initialization - захват ресурса при инциализации.
Суть данной идиомы заключается в том, что в конструкторе класса мы выделяем некоторые ресурсы. А в деструкторе, освобождаем для них место. Например:
```
struct S
{
   S(int arrLength): arrLength {arrLength}
   {
      std::cout<<std::endl<<"allocating new memory for array";
      this->arr = new int[arrLength];
   }
   ~S()
   {
      std::cout<<std::endl<<"memory cleared";
      delete[] this->arr;
   }
   int arrLength;
   int* arr;
};
```

Использование битовых полей и объединений

Использовать битовые поля следует для оптимизации работы с памятью. В качестве примера рассмотрим структуру `DateTime`:
```
struct DateTime
{
   unsigned int year: 12;
   unsigned int month: 4;
   unsigned int date: 5;
   unsigned int hour: 5;
   unsigned int min: 6;
   unsigned int second: 6;
};
```

В контексте задачи, поле `min` должно лишь хранить целое значение в диапазоне [0,60]. То есть нам достаточно лишь 6 бит.

Если вместо выделения битового поля, использовать обычную переменную типа `int`. То целых 26 бит оставались бы неиспользованными - в них хранилось бы значение `0`.

Какими особенностями обладают перечисления с областью видимости

Перечисления с областью видимости - `enum class` - обладают собственной областью видимости. Это свойство уменьшает риск возникновения ошибок вследствие одинаковых названий различных сущностей.