WEEK 11

Derived classes and resources + Standards

OVERVIEW

- Week 11-1
  - Constructor & Destructors Redux
  - Copy Constructor & Copy Assignment Operator Redux
- Week 11-2
  - Language Standards

CONSTRUCTORS

- A Default No Arg Constructor is provided by the compiler if not explicitly created
- Can have one or many constructors to create objects with
- A derived class' constructor will call its base's default constructor by default
- This can be overridden by explicitly calling a particular base constructor

DESTRUCTORS

- Default is provided by the compiler if not explicitly created
- Only one destructor that cleans up the object
- A derived class' destructor will call its base destructor automatically

COPY CONSTRUCTOR

- Default is provided by the compiler if not explicitly created
- The default CC only does shallow copying

COPY ASSIGNMENT OPERATOR

- Default is provided by the compiler if not explicitly created
- The default CA only does shallow copying

DEFAULT CC / CA

Consider the following class:

```
class MyClass {
int x;
char c;
std::string s;
};
```

What do the default copy constructor and copy assignment operator look like?

DEFAULT CC

Consider the following class:

```
class MyClass {
int x;
char c;
std::string s;
};
```

```
MyClass::MyClass(const MyClass& other) : x(other.x), c(other.c), s(other.s)
{}
```

A default copy constructor provided by the compiler ends up looking like this. Notice how the variables are passed

This is known as an initializer list

DEFAULT CA

Consider the following class:

```
class MyClass {
int x;
char c;
std::string s;
};
```

```
MyClass& MyClass::operator=( const MyClass& other )
{
    x = other.x;
    c = other.c;
    s = other.s;
    return *this;
}
```

This is what the default copy assignment provided by the compiler would look like

CC / CA

- In the case of compiler provided CC and CA, they have no notion of handling dynamic resources (deep copying). For this we would need to deal in a manually defined CC / CA
- How does this notion carry into inheritance / derived classes with resources?

DERIVED CLASSES WITH RESOURCES

Consider the following:

```
class Compound{
        int weight;
        char* name; // Dynamic
Resources
        …
}
```

```
class Playdoh : public Compound{
        char* colour; // Dynamic Resources
        …
}
```

Are there any considerations for implementing the CC / CA in this context?

COMPLAY EXAMPLE

LANGUAGE STANDARDS

INLINE FUNCTIONS

- Inlining is a technique introduced in C++11 to improve execution time of function calls by replacing the call itself with the function logic.
- This reduces the overhead associated with passing parameters.
- The trade off for this an increase in executable code.

INLINE FUNCTIONS

- We can make a request to the compiler that a function should be inlined at every call of that function.
- The best candidates for inlining are member functions that are short code blocks
- In the end however, the compiler will decide if it is more efficient to inline your function or not

INLINE FUNCTIONS

Inline method 1

```
// inline_1.h
 const int NG = 20;
 struct Student {
   private:
     int no;
     float grade[NG];
     int ng;
   public:
     void set(int n, const char* g);
     const float* getGrades() const {
         return grade;
     }
 };
```

The first method of inlining is to define a query in the header file as a one line return statement.

This is done within the class.

INLINE FUNCTIONS

The second method of inlining is to use the inline keyword.

Notice that this is outside of the class definition.

Inline method 2

```
// inline_2.h
const int NG = 20;
struct Student {
public:
     void set(int n, const char* g);
     const float* getGrades() const;
 };
inline const float* Student::getGrades() const
{ return grade; }
```

FUNCTION DELETION

- Assigning a function to the delete keyword will make it so that any attempt to implement the function (ie provide it definition) will cause compilation errors
- This is very useful to deny certain operations such as the copying of objects
  - Such as deleting the copy constructor and copy assignment

FUNCTION DELETION

Legacy

```
class Student {
    int no;
    float* grade;
    int ng;
    Student(const Student& source);
    Student& operator=(const Student& source);

public:
    Student();
    Student(int, const float*);
    ~Student();
    void display() const;
};
```

What do you notice about these functions?

FUNCTION DELETION

C++11

```
class Student {
    int no;
    float* grade;
    int ng;
public:
    Student();
    Student(int, const float*);
    ~Student();
    void display() const;
    Student(const Student& source) = delete;
    Student& operator=(const Student& source) = delete;
};
```

Delete here and here instead.

CASTING

C-Style

```
hours = (double) minutes / 60;  // C-Style
Cast
```

Function-Style

```
hours = double(minutes) / 60;  //
Function-Style Cast
```

Constrained Cast

```
hours = static_cast<double>(minutes) / 60;
```

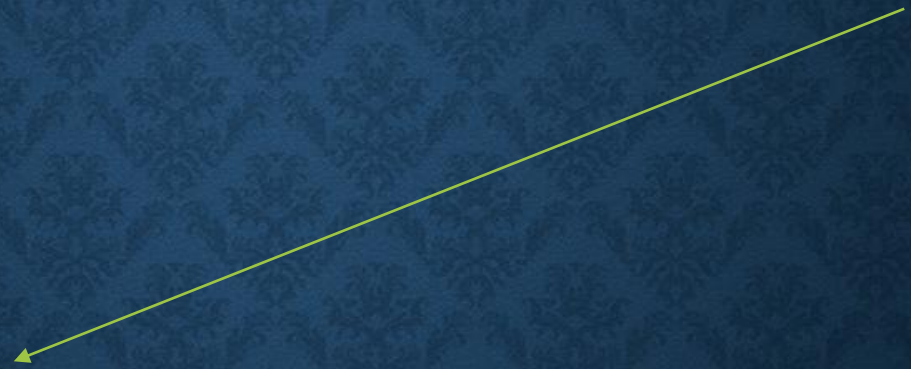
STD:NOTHROW

- In C++98 exception handling for dynamic memory allocation was added to the standard. By default the new operator would throw an exception if the operator encountered an error.
- Prior to C++98 the default was that the new operator would return null instead if it encountered an error (e.g. insufficient memory).
- The nothrow keyword was added to the standard in C++98 to allow for the pre C++98 behavior if desired instead of throwing an exception.

STD::NOTHROW

Pre-C++98

```
#include <iostream.h>
 int main() {
     char* p;
     int i = 0;
     do {
         p = new char[100001];
         i++;
     } while (p != NULL);
     cout << "Out of space after " << i << " attempts!\n";
 }
```



Would return null if not successful

STD::NOTHROW

Post-C++98

```
#include <new>
#include <iostream>
int main() {
    char* p;
    int i = 0;
    do {
        p = new (std::nothrow) char[100001];
        i++;
    } while (p != nullptr);

    std::cout << "Out of space after " << i << " attempts!\n";
}
```

Allows for the pre C++98 behavior rather than an exception