DIGITAL ARTS & ENTERTAINMENT

Copy Semantics

"Rule of three"

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Intro: Helper on function parameters

- Parameters can be divided into two different groups, depending in their usage:
- Input Parameters:
  - These parameters are used in the function. Their optional change of value has no effect on the state of a program or class.
  - Examples:

```
int
const int
const int&
const ClassName*
```

Intro: Helper on function parameters

- Parameters can be divided into two different groups, depending in their usage:
- Output Parameters:
  - These are passed to store the result of the function. Changing their value has an effect on the state of a program.
  - Examples:

```
int&
ClassName*
```

CppCoreGuidelines: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#f15-prefer-simple-and-conventional-ways-of-passing-information

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Copy semantics

- Destructor
- Copy assignment operator
- Copy constructor
- = delete and = default
- explicit constructor

Heap objects - assignment

```
struct Rectf
{
  float x, y, width, height;
}
```

What happens when a pointer is assigned to another pointer?

```
Rectf *pRect1 { new Rectf{} };
Rectf *pRect2 { nullptr };
pRect2 = pRect1;
```

Pointers - assignment

```
struct Rectf
{
  float x, y, width, height;
}
```

What happens when a pointer is assigned to another pointer?

```
Rectf *pRect1 { new Rectf{} }; // 1
Rectf *pRect2 { nullptr }; // 2
pRect2 = pRect1;
```

1. A new MyRect object is created, and its address is copied to pRect1.
2. The value of the pointer pRect1 is copied to the pointer pRect2.

There is no second MyRect object!

Objects - assignment

```
struct Rectf
{
  float x, y, width, height;
}
```

What happens when an object is assigned to another object?

```
Rectf rect1{10,10,100,100}, rect2{};
rect2 = rect1;
```

Objects - assignment

```
struct Rectf
{
  float x, y, width, height;
}
```

What happens when an object is assigned to another object?

```
Rectf rect1{10,10,100,100}, rect2{};
rect2 = rect1;
```

- The members of object rect1 are copied to rect2: “member wise copy” or shallow copy:

```
rect2.x = rect1.x;
rect2.y = rect1.y;
//etc...
```

This is the default behavior of classes and structs in C/C++.

default assignment operator

"operator="

- In C++, the compiler automatically generates the copy assignment operator.
- The default implementation copies all the data members. This is called a shallow copy.
- It makes simple user-defined types in C++ behave like structures do in C.

- This can sometimes lead to unexpected behaviour!

default assignment operator

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
private:
  unsigned int m_Size;
  int* m_pData;
};
```

```
Blob::Blob(int size)
  :m_Size{ size }
  ,m_pData{new int[size]}
{ }
Blob::~Blob()
{
  delete[] m_pData;
}
```

```
int main()
{
  Blob b1{ 10 };
  Blob b2{ 50 };
  b2 = b1;
}
```

The members of object b1 are copied to b2: "member wise copy" or shallow copy:

```
b2.m_Size  = b1.m_Size;
b2.m_pData = b1.m_pData;
```

What is the problem?

- The m_pData pointer is copied, both Blob objects point at the same dynamically allocated array.
- Possible Memory leak: the pointer b2.m_pData is overwritten.
- Double free error when the objects are destroyed.

DIGITAL ARTS & ENTERTAINMENT

define the assignment operator

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs);
private:
  unsigned int m_Size;
  int* m_pData;
};
```

```
Blob & Blob::operator=(const Blob & rhs)
{
  if(&rhs != this){
    m_Size = rhs.m_Size;
    delete[] m_pData;
    m_pData = new int[rhs.m_Size];
    for (int i = 0; i < m_Size; ++i)
    {
      m_pData[i] = rhs.m_pData[i];
    }
  }
  return *this;
}
```

Solution 1: define the operator=

- Check for self-assignment (!)
- Copy non-pointer members
- Delete the old heap object
- Create a new heap object using new size
- Copy the data from the rhs argument
  - "DEEP COPY"
- Return the object to allow "chaining".

```
int main()
{
  Blob b1{ 10 };
  Blob b2{ 50 };
  b2 = b1;
}
```

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

delete the assignment operator

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs) = delete;
private:
  unsigned int m_Size;
  int* m_pData;
};
```

Solution 2: delete the operator=

- it is now not possible to use the copy assignment operator.
- Using the assign operator results in a compile error.

```
Blob & Blob::operator=(const Blob & rhs)
{
  if(&rhs != this){
    m_Size = rhs.m_Size;
    delete[] m_pData;
    m_pData = new int[rhs.m_Size];
    for (int i = 0; i < m_Size; ++i)
    {
      m_pData[i] = rhs.m_pData[i];
    }
  }
  return *this;
}
```

```
error C2280: 'Blob &Blob::operator =(const Blob &)':
attempting to reference a deleted function
```

```
int main()
{
  Blob b1{ 10 };
  Blob b2{ 50 };
  b2 = b1;
}
```

the assignment operator

Conclusion:

- When a class uses heap objects, it needs a destructor.
- When a class has a destructor it also needs an explicit defined implemented or deleted assignment operator
- So, if there is a destructor, there MUST be an assignment operator.

- This assumes the destructor is deleting resources.
- Avoid empty destructors by simply not declaring them.

DIGITAL ARTS & ENTERTAINMENT

the assignment operator

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs) = delete;
private:
  unsigned int m_Size;
  int* m_pData;
};
```

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs);
private:
  unsigned int m_Size;
  int* m_pData;
};
```

```
class Player final
{
public:
  Player(const std::string& name);

private:
  const std::string& m_Name;
  int m_Score;
};
```

```
class Player
{
public:
  Player(const std::string& name);
  virtual ~Player() = default;
private:
  const std::string& m_Name;
  int m_Score;
};
```

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Automatic generation of methods by the compiler

In C++, the compiler automatically generates these methods for a class if it does not declare it on its own (there are exceptions, see later):

- default constructor
- destructor
- copy-assignment operator
- copy constructor

The copy constructor

```
Blob b1; // 1
Blob b2{b1}; // 2
Blob b3 = b1; // 3
```

What happens when an object is created using another object as parameter?

1. A new Blob object b1 is created on the stack.
2. New Blob object named b2 is created using the automatically generated copy constructor.
3. New Blob object named b3 is created using the automatically generated copy constructor.
   - ➢ All members from b1 are copied to b2 and b3

DIGITAL ARTS & ENTERTAINMENT

The copy constructor

```
Blob b1; // 1
Blob b2{b1}; // 2
Blob b3 = b1; // 3
```

What happens when an object is created using another object as parameter?

1. A new Blob object b1 is created on the stack.
2. New Blob object named b2 is created using the automatically generated copy constructor.
3. New Blob object named b3 is created using the automatically generated copy constructor.
   - ➢ All members from b1 are copied to b2 and b3

- ➢ This is the default behavior of classes and structs in C/C++.

The copy constructor

- The same problem can happen here, for the same reason.
  - What if one of the data members is a pointer?
    - the pointer value is copied to the other object using pass by value of the object in a method.
    - both objects have a pointer pointing at the same heap object.
    - what if one of them is destroyed, deleting the heap object?
    - Well, the other has no idea that the heap object no longer exists, resulting in a dangling pointer hat could lead to a crash!

Avoid the automatic generation of the copy constructor by explicitly declaring or deleting it when default behavior is not desired.

Example implementation of the copy constructor

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs);
  Blob(const Blob& other);
private:
  unsigned int m_Size;
  int* m_pData;
};
```

Declaration of the copy constructor in the header file.

Example implementation of the copy constructor

const reference to the existing object that will be copied.

```
Blob::Blob(const Blob & other)
  :m_Size{ other.m_Size }
  ,m_pData{new int[other.m_Size]}
{
  for (int i = 0; i < m_Size; ++i)
  {
    m_pData[i] = other.m_pData[i];
  }
}
```

Example implementation of the copy constructor

```
Blob::Blob(const Blob & other)
  :m_Size{ other.m_Size }
  ,m_pData{new int[other.m_Size]}
{
  for (int i = 0; i < m_Size; ++i)
  {
    m_pData[i] = other.m_pData[i];
  }
}
```

Memberwise copy of all the POD (Plain Old Data) members (integer, float, double, bool) from the argument object to "this" object.

Example implementation of the copy constructor

```
Blob::Blob(const Blob & other)
  :m_Size{ other.m_Size }
  ,m_pData{new int[other.m_Size]}
{
  for (int i = 0; i < m_Size; ++i)
  {
    m_pData[i] = other.m_pData[i];
  }
}
```

A new heap object is created, its memory address is stored in the pointer m_pData.

Example implementation of the copy constructor

```
Blob::Blob(const Blob & other)
  :m_Size{ other.m_Size }
  ,m_pData{new int[other.m_Size]}
{
  for (int i = 0; i < m_Size; ++i)
  {
    m_pData[i] = other.m_pData[i];
  }
}
```

Data is copied from the other object to this object

Prevent the generation of the copy constructor

Objects of this class can no longer be copy - constructed.

```
class Blob final
{
public:
  Blob(int size);
  ~Blob();
  Blob& operator=(const Blob& rhs) = delete;
  Blob(const Blob& other) = delete;
private:
  unsigned int m_Size;
  int* m_pData;
};
```

```
int main()
{
  Blob b1{ 10 };
  Blob b2{ b1 };
}
```

```
error C2280: 'Blob::Blob(const Blob &)': attempting to reference a
deleted function
```

default assignment operator

Conclusion:

- When a class uses dynamically allocated memory, it requires:
  1. a defined destructor.
  2. a defined or deleted assignment operator
  3. a defined or deleted copy constructor.

Also known as: "the rule of three"

Assignment operator vs copy constructor

- The assignment operator is used to copy the values from one object to another *already existing object*. The key words here are "already existing". Example:

```
MyRect r1, r2(20,50);
r1 = r2;
```

- A copy constructor is a special constructor that initializes a *new object* from an existing object. Example:

```
MyRect r1{10,20};
MyRect r2{r1};
MyRect r3 = r1;
```

Summary

- The class has an empty destructor (creates no heap objects) :
  - Do not write the destructor, copy assignment operator and copy constructor. They are automatically generated.
- The class has a nonempty destructor that deletes objects, two options:
  - Or disable the use of the default assignment operator and default copy constructor by inhibiting their automatic generation: delete them
  - Or implement both definitions.

Rule of three

- "The rule of three (also known as the Law of The Big Three or The Big Three is a rule of thumb in C++ (prior to C++11) that claims that if a class defines one of the following it should probably explicitly define all three."
- "These three functions are special member functions. If one of these functions is used without first being declared by the programmer, it will be implicitly implemented by the compiler with the default semantics of performing the said operation on all the members of the class."(wiki)
- Rule of the Big Three:
  - destructor
  - copy constructor
  - assignment operator

```
~T();
T( const T& );
T& operator=( const T& );
```

DIGITAL ARTS & ENTERTAINMENT

Never define an empty destructor function

```
class Bullet final
{
public:
  Bullet(int size);
  ~Bullet();
private:
  const int m_Size;
};
```

```
Bullet::Bullet(int size)
   : m_Size {size}
{

};

Bullet::~Bullet()
{

};
```

- This is confusing
  - The header contains a destructor declaration but has not the other two declarations
    - Assuming its bad code, not following the rule of three, but its ok after looking at the definition of the function.

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

DIGITAL ARTS & ENTERTAINMENT

Never define an empty destructor function

```
class Bullet final
{
public:
  Bullet(int size);
  ~Bullet() = default;
private:
  const int m_Size;
};
```

```
Bullet::Bullet(int size)
    : m_Size {size}
{

};
```

- add the = default syntax instead

DIGITAL ARTS & ENTERTAINMENT

Never define an empty destructor function

```
class Bullet
{
public:
  Bullet(int size);
  virtual ~Bullet() = default;
private:
  const int m_Size;
};
```

```
Bullet::Bullet(int size)
   : m_Size {size}
{

};
```

- In case of a virtual base class destructor, add the = default syntax

Shallow vs deep copy

- Shallow copy or memberwise copy:
  - is what the generated copy constructor and default assignment operator do
  - all members are copied
    - the pointer values are copied, not the objects they point to (!)
- Deep:
  - own implementation of copy constructor and assignment operator
  - all objects are duplicated
  - all members are copied

Special case: What if there are const members?

```
class Bullet final
{
public:
  Bullet(int size);
private:
  const int m_Size;
};
```

```
Bullet::Bullet(int size)
   : m_Size {size}
{};
```

Compiler generates:

- Deleted copy assignment operator
- Default copy constructor

- A class has non-static data members that cannot be copied, in other words: const data members.
- Ok, lets recap:
  - An implicitly generated copy assignment operator copies all data members.
  - If a member is const it can not be modified after initialization. The copy operation would fail.
  - That is why in that case an implicit deleted copy assignment function is generated.
  - It is however still possible the write an explicit copy assignment operator!

About declaring functions as default

```
class Bullet final
{
public:
 Bullet(int size);
 ~Bullet() = default;
 Bullet(const Bullet& other) = default;
 Bullet& operator=(const Bullet & rhs) = default;
private:
  int m_Size;
};
```

implicitly defined, has an empty body

perform full member-wise copy of the object's bases and non-static members, in their initialization order, using direct initialization.

When forced to be generated, no function definitions in the cpp file are allowed.

Same effect as not writing them at all.

About declaring functions as default

```
class Bullet final
{
public:
 Bullet(int size);
 ~Bullet() = default;
 Bullet(const Bullet& other) = default;
 Bullet& operator=(const Bullet & rhs) = default;
private:
  int m_Size;
};
```

```
class Bullet final
{
public:
 Bullet(int size);
private:
  int m_Size;
};
```

Both have an identical result

About declaring functions as default

- The destructor, copy constructor and copy assignment operator can be defined as
  - deleted
  - default
- deleted: it is not possible to use these functions
- default: no explicit function definition in the cpp file is allowed
  - destructor has an empty body
  - the copy functions perform full member-wise copy of the object's bases and non-static members, in their initialization order, using direct initialization.

References

- http://www.learncpp.com/
- https://msdn.microsoft.com/nl-be/library/dn457344.aspx
- *The C++ Programming Language* (3 ed.). Addison-Wesley. 2000. pp. 283–4. ISBN 978-0-201-70073-2