Object-Oriented Programming

Lesson 5

References
Copy constructor

References

A reference (&) is like a constant pointer that is automatically dereferenced.

There are certain rules when using references:

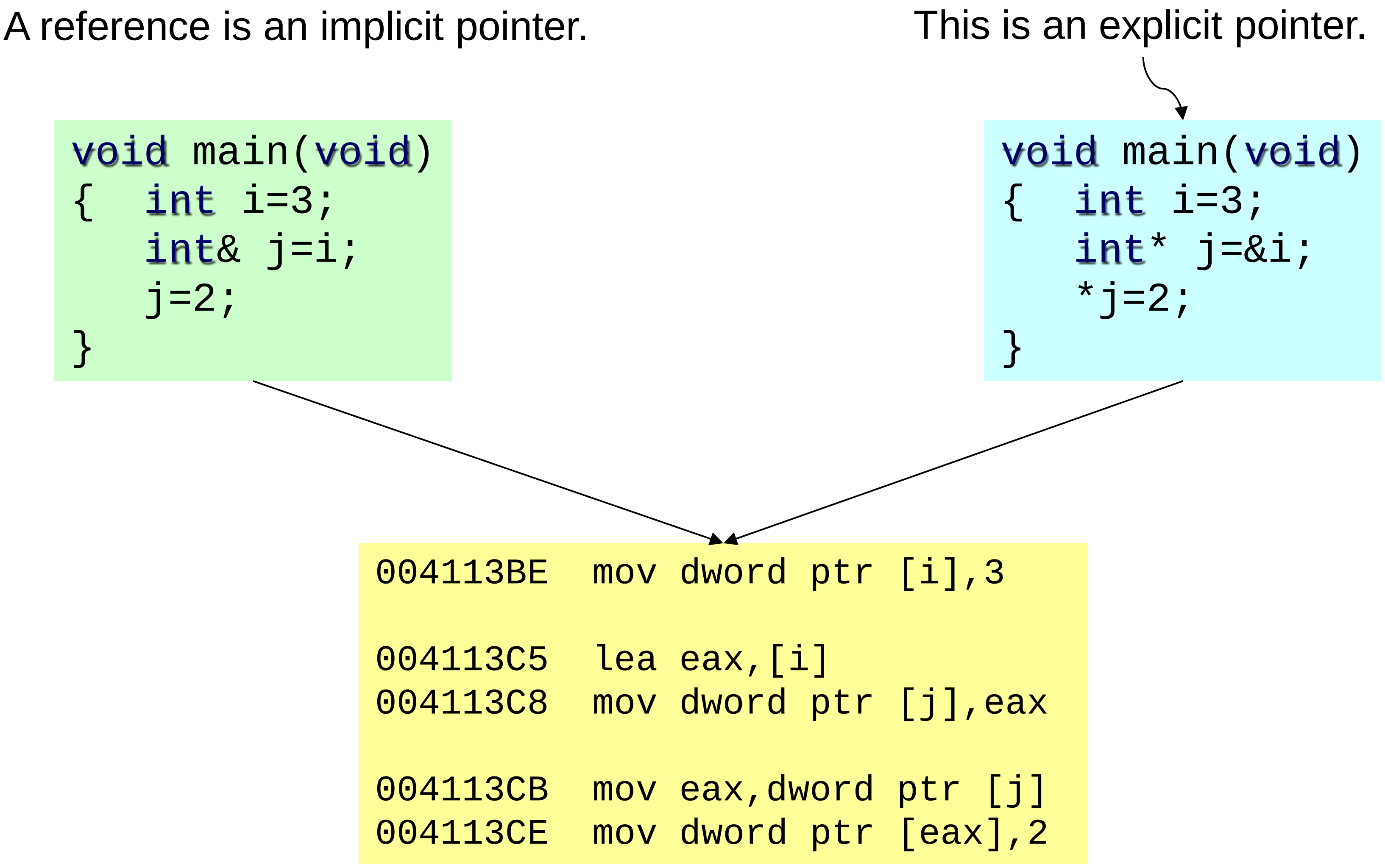
1. A reference must be initialized when it is created (pointers can be initialized at any time).
2. Once a reference is initialized to an object, it cannot be changed to refer to another object (pointers can be pointed to another object at any time).
3. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

```
int a = 47;
int* pa = &a;
*pa = 10;
```

```
int& ra = a;
ra = 13;
int& test; //error
```

In a declaration, T& means reference to an object of type T.

References



A reference is an implicit pointer.

```
void main(void)
{   int i=3;
    int& j=i;
    j=2;
}
```

This is an explicit pointer.

```
void main(void)
{   int i=3;
    int* j=&i;
    *j=2;
}
```

```
004113BE  mov dword ptr [i],3

004113C5  lea eax,[i]
004113C8  mov dword ptr [j],eax

004113CB  mov eax,dword ptr [j]
004113CE  mov dword ptr [eax],2
```

lvalue and rvalue

lvalue – variable on the left-hand side in an assignment operator.

rvalue – constant, variable or expression appearing on the right-hand side in an assignment operator.

Array identifier is not an lvalue; you cannot assign to it.

```
int main()
{
        int a[3] = { 0, 1, 2 };
        a = { 1, 2, 3 }; // error
}
```

Reference is an address and can therefore be used as lvalue.

References

A reference can be returned from a function.

In this case the function can be used as lvalue.

```
int F(int& i) { return i; }

int& RF(int& j) { return j; }

void main(void)
{
       int x=3;
       F(x) = 6;     // error
        RF(x) = 6;   // Ok
}
```

References

```
int* f (int* x)
{
        (*x)++;
        return x;
}

int& g (int& x)
{       x++;
        return x;
}

int main()
{
        int a = 0;
        f(&a);
        g(a);
}
```

References are frequently used in function argument lists.

When a reference is used as a function argument, any modification to the reference *inside* the function will cause changes to the argument *outside* the function.

If you return a reference from a function, you must take the same care as if you return a pointer from a function. Whatever the reference is connected to shouldn’t go away when the function returns, otherwise you’ll be referring to unknown memory.

References

The use of **const** references in function arguments is especially important because your function may receive a temporary object. This might have been created as a return value of another function or explicitly by the user of your function. Temporary objects are always **const**, so if you don’t use a **const** reference, that argument won’t be accepted by the compiler.

```
void f (int&) {}

void g (const int&) {}

void p (int*) {}

int main()
{
    // f (1);
    g (1);
    // p (1);
}
```

References

Your normal habit when passing an argument to a function should be to pass by const reference!

1) To pass an argument by value requires a constructor call, but if you're not going to modify the argument then passing by const reference only needs an address pushed on the stack.
2) There is a guarantee that the function will not modify the object ⇒ service for the class user.
3) The syntax of calling the function is identical to that pf pass-by-value ⇒ service for the class user.
4) It is possible to pass temporary objects.

Copy constructor

```
class CBook
{
        char* m_sTitle;
        unsigned m_nYear;
public:
        CBook (char* title, unsigned year);
        virtual ~CBook();
};
```

```
CBook goodBook = CBook("C++", 2014);
CBook anotherBook = goodBook;
```

```
CBook::CBook(char* title, unsigned year)
{       if (title == 0) m_sTitle = 0;
        else
        {       unsigned len = strlen(title) + 1;
                m_sTitle = new char [len];
                strcpy_s (m_sTitle,  len, title);
        }
        m_nYear = year;
}
CBook::~CBook()
{       delete [] m_sTitle;  }
```

Copy constructor

When create a new object from an existing object, a special function is called – the copy constructor.

New objects are created from the existing objects when:

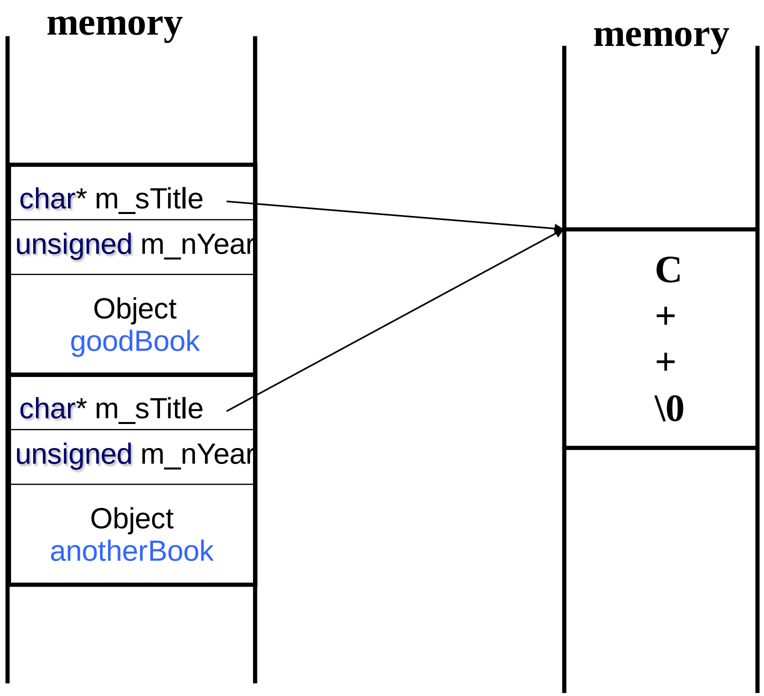
- you pass an object by value (you create a new object, the passed object inside the function frame, from an existing object, the original object outside the function frame);
- you return an object from a function;
- you explicitly assign one object to a new object of the same type.

If you do not implement a copy constructor this will be synthesized by the compiler.

The copy constructor synthesized by the compiler makes a simple bitcopy of the excisting object.
However, a bitcopy not makes sense, because it doesn’t necessarily implement the proper meaning.

Copy constructor

```
CBook goodBook = CBook("C++", 2014);
CBook anotherBook = goodBook;
```



A problem appears when the objects goodBook and anotherBook go out of scope (and need to be destroyed)

The first object to be destroyed is anotherBook. Its desctructor will be called and will release storage occuppied by the book's title.

Afterwards the object goodBook will be destroyed and its destructor will try to release storage occuppied by the book's title, which has already been released by anotherBook destructor!

Copy constructor

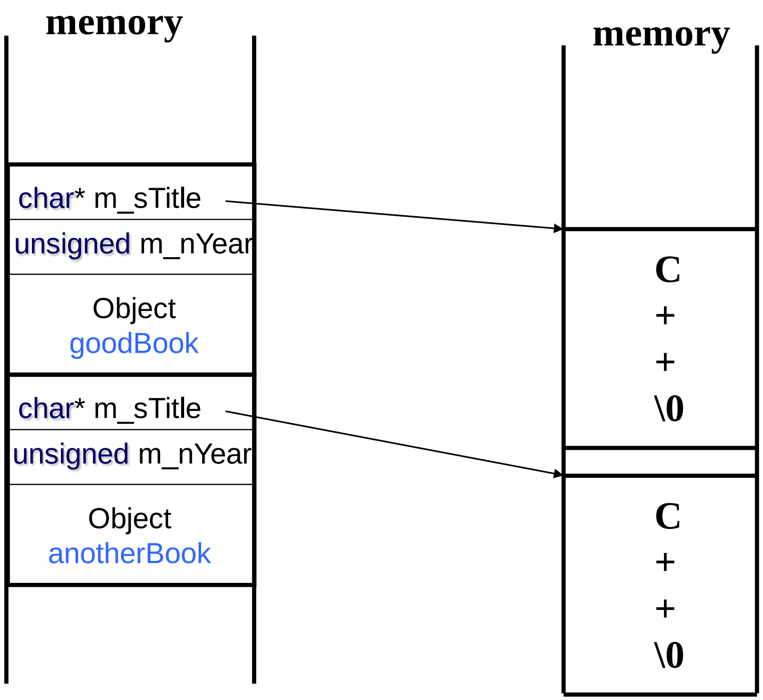
If your class uses dynamic memory allocation, you should always implement the proper copy constructor!

A copy constructor always receives a const reference to an object of the same class.

```
CBook::CBook(const CBook& Book)
{
        if (Book.m_sTitle == 0)
                m_sTitle = 0;
        else
        {
                unsigned len = strlen(Book.m_sTitle) + 1;
                m_sTitle = new char [len];
                strcpy_s (m_sTitle,  len, Book.m_sTitle);
        }
        m_nYear = Book.m_nYear;
}
```

Copy constructor

```
CBook goodBook = CBook("C++", 2014);
CBook anotherBook = goodBook;
```



```
CBook::CBook(const CBook& Book)
{
   if (Book.m_sTitle == 0)
          m_sTitle = 0;
   else
   {
          unsigned len = strlen(Book.m_sTitle) + 1;
          m_sTitle = new char [len];
          strcpy_s (m_sTitle,  len, Book.m_sTitle);
   }
   m_nYear = Book.m_nYear;
}
```

Each object is destroyed in a correct manner!

Copy constructor

Copy constructor is also called when you pass an object by value.

```
void compare (CBook a, CBook b)
{
   if ( a.m_nYear() == b.
       (strcmp(a.m_sTitle
        cout << " equal" 
   else
        cout << "not equa
}
```

```
int main(int argc, char* argv[])
{
   CBook Book1 = CBook("C++", 2014);
   CBook Book2 = Book1;
   CBook Book3 = CBook("Java", 2015);

   compare(Book1, Book2);
   compare(Book1, Book3);
   return 0;
}
```

The function `compare` receives as arguments two objects a and b by value. These objects will be created on the function’s stack. When the function terminate, all local objects have to be destroyed from the stack. If the copy constructor is not implemented, than the same problem as before will appear.

Copy constructor

A copy constructor is called upon:

- construction of a new object from the existing object:

```
type new_item = type (old_item);
```

- pass-by-value:

```
void function (type);
```

- return-by-value:

```
type function ();
```

The return optimization

```
const type type::f ()
{
        type tmp (/*arguments*/);
        return tmp;
}
```

1. Constructor for `tmp`
2. Copy constructor
3. Destructor of `tmp`

```
const type type::f ()
{
        return type(/*arguments*/);
}
```

1. Constructor

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapter 11