Xilinx Zynq FPGA,TI DSP, MCU 기반의 프로그래밍 전문가 과정

날 짜 : 2018 . 5. 30

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

< 1. friend with Classes >

-C 에서 구조체 포인터 배열이 있었던 것처럼 C++ 은 Class 포인터 배열이 존재 한다.

```
#include<iostream>
using namespace std;

class Counter
{
	int val;
	public:
		Counter(void)
		{
			val=0;
		}
		void Print(void)
		{
			cout << val << endl;
		}
		friend void SetVal(Counter& C, int val);
};

void SetVal(Counter& c, int val)
{
	c.val = val;
}

int main(void)
{
	Counter cnt;
	cnt.Print();
	SetVal(cnt, 2002);
	cnt.Print();
	return 0;
}
```

---

https://blog.naver.com/star7sss/220826110723

< 2. 한쪽만 Friend 인 경우 >

```
#include<iostream>
using namespace std;

class A
{
	private:
		int data;
		friend class B;
};

class B
{
	public:
		void SetData(A& a, int data)
		{
			a.data = data;
		}
};

int main(void)
{
	A a;
	B b;
	b.SetData(a,10);
	return 0;
}
```

- A클래스가 B클래스에게 친구권한을 주었기 때문에
  A의 private int data에 접근이 가능하다.

- B클래스는 A에게 아무 권한도 주지 않았다. friend가 아니다.
  따라서 접근이 불가능 하다.

< 3. 복사생성자 copy constructor >

```
#include<iostream>
using namespace std;

class A
{
	public:
		A(void)		// 첫번째꺼를 받음.
		{
			cout << "A()Call"<< endl;
		}
		A(int i)		// 두번째꺼를 받음.
		{
			cout << "A(int i)) Call" << endl;
		}
		A(const A& a)	// 복사 생성자
		{
			cout <<"A(const A& a) Call" <<endl;
		}
}
;

int main (void)
{
	A obj1;
	A obj2(10);
	A obj3(obj2);
	return 0;
}
```

---

1. 기본생성자(디폴트 생성자)
2. 인자가 있는 생성자
3. 인자로 자신의 타입의 객체를 가지는 생성자

(C언어로 생각하면 자기 참조를 하는 구조체 A로 볼 수 있다.
위의 obj2는 타입이 클래스 형태로 되어 있다.
즉, **객체를 생성자로 받을 때는 const A &a 라는 형식**으로 받게 된다.)

< 4. 복사생성자, 멤버 변수도 같이 복사하는 경우 >

```
#include<iostream>
using namespace std;

class Point
{
	int x, y;
	public:
	Point(int _x, int _y)
	{
		x = _x;
		y = _y;
	}
	void ShowData(void)
	{
		cout << x << ' ' << endl;
		cout << y << ' ' << endl;
	}
}
;

int main (void)
{
	Point p1 (10, 20);
	Point p2 (p1);

	p1.ShowData();
	p2.ShowData();
	return 0;
}
```

---

http://thrillfighter.tistory.com/146
http://metalkim.tistory.com/245

얇은 복사
- 값만 복사하는 형태
- 포인터 등을 복사 시 같은 주소를 참조하기 때문에, 값 변경, 동적 해제 등에서 문제가 발생한다.
(한 대상의 값을 변경하면 다른 대상의 값 또한 바뀌어 버리는 등의 문제)

깊은 복사
- 주소가 가리키는 값. 즉, 그 메모리의 값을 복사하는 복사 형태.
- 포인터 등을 복사 시 한 주소를 참조하지 않고 , 새로 주소를 할당받은 후 참조하기 때문에, 문제가 발생하지 않는다.

< 5. 복사생성자2 >

```
#include<iostream>
#include<string.h>
using namespace std;

class Person
{
	char *name;
	char *phone;

	public:
		Person(char* _name, char *_phone);
		Person(const Person& p);
		~Person();
		void ShowData();
};

Person::Person(char *_name, char *_phone)
{
	name = new char[strlen(_name) +1];
	strcpy(name, _name);
	phone = new char[strlen(_phone) + 1];
	strcpy(phone, _phone);
}

Person::~Person(void)
{
	delete []name;
	delete []phone;
}

Person::Person(const Person& p)
{
	name = new char[strlen(p.name) +1];
	strcpy(name, p.name);
	phone = new char[strlen(p.phone) +1];
	strcpy(phone, p.phone);
}

void Person::ShowData(void)
{
	cout <<"name:" << name <<endl;
	cout << "phone:" << phone << endl;
}
```

```
int main (void)
{
	Person p1("Jo", "011-9272-6523");
	Person p2 =p1;

	return 0;
}
```

- segment fault 를 피하기 위해서 복사 생성자를 사용한다.
- Person::Person(const Person& p) 부분을 추가해 주어야 같은 메모리 공간을 p1, p2가 동시에 가리키지 않게 되어 Segmentation Fault 를 막을 수 있다.
(위에서 언급한 **얕은 복사**의 형태에 해당한다.)

< 6. 복사생성자3 >

```
#include<iostream>
using namespace std;

class A
{
	int val;
	public:
		A(int i)
		{
			cout << "A(int i)Call" << endl;
			val = i;
		}
		A(const A& a)
		{
			cout << "A(const A& a) Call" << endl;
			val = a.val;
		}
		void ShowData(void)
		{
			cout << "val:" << val << endl;
		}
};
```

```
void function(A a)
{
	a.ShowData();
}

int main(void)
{
	A obj(30);
	function(obj);
	return 0;
}
```

```
#include<iostream>
using namespace std;

class A
{
	int val;
	public:
		A(int i)
		{
			cout<< "A(int i) Call" << endl;
			val = i;
		}
		A(const A& a)
		{
			cout << "const A& a) Call " <<endl;
			val = a.val;
		}
		void ShowData(void)
		{
			cout << "val:" << val <<endl;
		}
};
A function(A& a)
{

	return a;
}

int main(void)
{
	A a(10);
	function(a).ShowData();
	return 0;
}
```

---

* 복사 생성자는 언제 동작하는가?
1. 기존에 생성된 객체로 새로운 객체를 초기화 할 때
2. 함수 호출 시 객체를 Reference 가 아닌 형태로 전달할 경우.
3. 함수 내에서 객체를 Reference 가 아닌 형태로 return 하는 경우가 있다.

< 7. 상속 >

```
#include<iostream>
#include<string.h>
using namespace std;

class Person
{
	int age;
	char name[20];
	public:
		int GetAge(void)const		// 함수 오버로딩을 할때 써먹는 const
		{
			return age;
		}
		const char *GetName(void) const
		{
			return name;
		}
		Person(int _age = 1, char *_name = "noname")
		{
			age = _age;
			strcpy(name, _name);
		}
};

class Student: public Person
{
	char major[20];
	public:
		Student(char *_major)
		{
			strcpy(major, _major);
		}
		const char *GetMajor(void) const
		{
			return major;
		}
		void ShowData(void) const
```

```
        {
            cout << "name: " << GetName() << endl;
            cout << "age:" << GetAge() << endl;
            cout << "major:" << GetMajor() << endl;
        }

};

int main(void)
{
    Student Park("Computer Science");
    Park.ShowData();
    return 0;
}
```

---

1. 여러번 사용할 수 있게 하기 위해 상송을 사용한다.
2. 이름을 바꾸는 등의 플랫폼만 변화 시켜 쓸 수 있다.