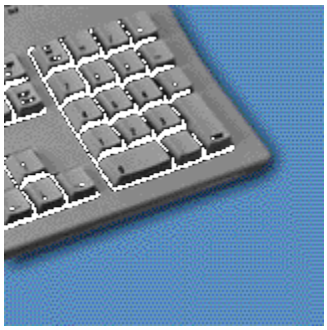


2. C++의 주제들 I

<장도비라>
이전 장에서 우리는 일반적인 윈도우 프로그래밍의 구조를 살펴보았다. 후에 우리는 이전 장에서 작성한 간단한 윈도우 프로그램을 MFC의 구조처럼 만들 것이다. 그에 앞서 이 장부터 우리는 MFC의 구조 설계에 필요한 C++의 개념들을 살펴보기 시작한다. C++에 익숙한 독자들도 2,3,4장의 주제들을 다시 한번 꼼꼼하게 짚어보고 복습하는 것이 MFC의 구조를 설계하는데 도움이 될 것이다. 이 장에서는 플래그, 함수포인터, 멤버 함수 참조 연산자, 핸들과 변환 연산자 오버로딩 등의 주제들을 살펴본다.
</장도비라>

우리는 이 장에서 아래의 내용들□을 살펴본다.

<저자 한마디>
필자는 독자들이 C++를 자유롭게 다룰 수 있다고 가정한다. 이 장과 4장에서 설명하는 내용은 모두 일반적인 C/C++의 주제들로써 이 책에서 사용되는 내용 중 설명이 필요하다고 생각된 부분만을 선택한 것이다. 필자는 각 주제들을 설명할 때, 초보자들에게 하듯 하지 않을 것이므로, 이 장과 4장에서 설명하는 내용의 70% 이상이 이해되지 않는 독자들은 인터넷 서점에서 평이 좋은 책을 골라, 먼저 C/C++ 관련 주제들을 공부하길 바란다.
</저자 한마디>

- 비트 플래그(bit flag)
- 함수 포인터(pointer to a function)

[[[[[모든 문장 및 소스에서 ->, >와 <등의 탈자가 발생하지 않도록 주의해 주세요. 예를 들면 아래는 .*, ->*입니다. 또한 메뉴의 단계를 나타내는 →와 C++의 멤버 참조 연산자 ->도 다르니 편집에 주의해 주세요]]]]]

- .*, ->* 연산자
- #와 ##
- 핸들(handle)
- 변환 연산자 오버로딩(conversion operator overloading)

비트 플래그(bit flag)

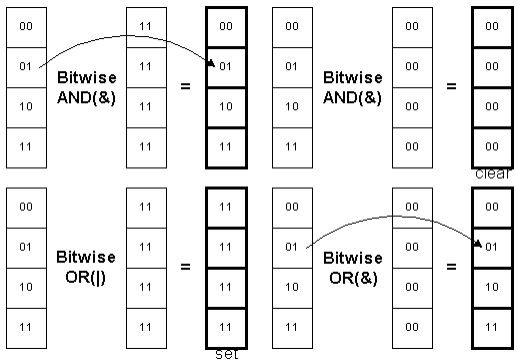
<절도비라>
이 절에서는 윈도우즈 API 함수에서 빈번하게 사용되는 비트 플래그의 동작과 원리에 대해서 살펴본다. 또한 비트 마스크를 이용하여 특정한 비트를 1이나 0으로 만드는 방법과 설정된 비트값을 얻기 위해 쉬프트 연산자를 이용하는 방법을 익힌다.
</절도비라>

특정한 상태의 기록을 나타내기 위해서 사용된 변수를 **깃발(Flag) 변수**라고 한다.

어떤 부호 없는 정수타입 변수(unsigned int)의 이진 비트열의 각각의 비트들이 이러한 깃발 변수로 사용되었을 때, 이를 **비트 플래그(bit flag)**라고 한다.

예를 들어 윈도우즈 Win32시스템에서 정수타입 변수는 32비트이므로 32가지의 On/Off 상태를 기록할 수 있다. 만약 상태의 개수가 각각 4가지라면, 상태를 나타내는데 2비트가 필요하므로 16가지의 깃발 변수로 사용할 수 있다. 각 비트의 상태를 변경하기 위해 비트 마스크(bit mask)를 사용한다.

비트 마스크(bit mask)는 비트 연산자 &, | 를 사용하여 특정한 비트를 1(bit set)로 혹은 0(bit clear)으로 만드는 것을 말한다. & 와 | 연산은 아래 [그림 2.1]과 같은 특징을 가진다.



[그림 2.1] 비트 마스크의 원리

즉, 1과 & 하면 원래의 비트가 유지되며, 0과 & 하면 모두가 지워진다. 또한 1과 | 하면 모두가 1이 되며, 0과 | 하면 원래의 비트가 유지된다.

예를 들면 MessageBox()함수가 버튼의 종류를 나타내기 위해, 최하위 4비트(3,2,1,0위치)를 사용 - 나타낼 수 있는 버튼의 종류는 16가지가 된다 - 하고, 아이콘의 종류를 나타내기 위해 그 다음 4비트(7,6,5,4위치)를 사용한다고 가정해 보자. 그러면 버튼의 종류와 아이콘의 종류를 다음 [예제 2.1]과 같이 나타낼 수 있다(예제를 보면 버튼의 종류가 6가지이므로, 3비트면 충분하지만, 확장성을 위해 4비트를 예약한 것 같다).

[예제 2.1] MessageBox()의 비트 플래그들

```
#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
#define MB_RETRYCANCEL              0x00000005L


#define MB_ICONHAND                 0x00000010L
#define MB_ICONQUESTION             0x00000020L
#define MB_ICONEXCLAMATION          0x00000030L
#define MB_ICONASTERISK             0x00000040L
```

이제 버튼의 종류와 아이콘의 종류를 나타내는 비트 플래그를 설정하기 위해 비트 OR연산자를 이용하여 아래처럼 설정할 수 있다.

MB_OK | MB_ICONQUESTION

또한, 설정된 비트 플래그 flag에서 아이콘의 상태를 얻어 내기 위해 아래처럼 사용할 수 있다.

iIconState = (flag & 0x00f0) >> 4;

비트 플래그의 사용을 보여주는 [예제 2.2]를 보자.

[예제 2.2] 비트 플래그의 적절한 사용

```
#include <stdio.h>

#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
#define MB_RETRYCANCEL              0x00000005L


#define MB_ICONHAND                 0x00000010L
#define MB_ICONQUESTION             0x00000020L
#define MB_ICONEXCLAMATION          0x00000030L
#define MB_ICONASTERISK             0x00000040L

void MessageBox(const char* pszMessage, unsigned int flag)
{
    int iButton = flag & 0x000f;
    int iIcon   = (flag & 0x00f0) >> 4;
    char*   pszButtons[] = { "OK", "OKCANCEL",
                             "ABORTRETRYIGNORE", "YESNOCANCEL",
                             "YESNO", "RETRYCANCEL" };
    char*   pszIcons[] = { "", "HAND", "QUESTION",
                           "EXCLAMATION", "ASTERISK" };

    printf( "msg = %s\n"
            "button = %s\n"
            "icon = %s\n",
            pszMessage, pszButtons[iButton], pszIcons[iIcon] );
}//MessageBox()

void main()
{
    MessageBox( "hello", MB_OK|MB_ICONQUESTION );
}//main()
```

출력 결과는 다음과 같다.

[[[[[편집 주의: 출력 결과는 '소스강조' 스타일을 적용했습니다]]]]]

```
msg = hello
```

```
button = OK
icon = QUESTION
```

비트 플래그는 부호없는 정수(unsigned int)의 각 비트 위치에 정보를 담는 것을 말한다. 앞으로 독자들은 비트 플래그를 파라미터로 전달 받는 많은 함수들을 보게 될 것이다.

함수 포인터(pointer to a function)

<절도비라>
이 절에서는 함수의 시작 주소를 가리키는 포인터 변수인 함수 포인터의 원리를 살펴본다. 또한 함수 포인터 변수를 선언하는 방법과 초기화하는 방법을 익히고, 함수 포인터를 이용하여 실행 시간에 호출할 함수를 바인딩(binding)하는 방법을 익힌다.
</절도비라>

함수 포인터란 함수의 시작 주소를 가리키는 포인터를 말한다. Win32환경에서 포인터는 4바이트며, 함수 포인터의 4바이트 내용은 특정 함수의 시작 주소를 담고 있다. 함수 포인터의 원리와 용도를 이해하는 것은 무척 중요하다.

함수 포인터를 이용하면 일반적인 함수(generic function)를 설계할 수 있다. 예를 들면 C의 표준함수 qsort()는 함수 포인터를 파라미터로 받아서, 어떠한 데이터 타입에 대한 소팅(sorting)도 수행할 수 있다. 우리는 함수 포인터를 일반적인 디자인(generic design)에 적용하는 예를 5장과 6장에서 살펴볼 것이다. 아래의 소스는 함수 포인터 변수를 선언하고, 이를 Sum()함수의 시작 주소로 초기화 한 다음, 함수 포인터를 통해 Sum()을 호출하는 방법을 보여준다.

```
#include <stdio.h>

int Sum(int a, int b)
{
    return a+b;
}//Sum()
```

```
void main()
{
    int (*fp)(int,int);
    int i;

    fp = Sum; // Sum()함수의 시작 주소를 fp에 대입한다.
    i = fp(2,3); // fp(2,3) == Sum(2,3)
    printf( "%d\n", i );
}//main()
```

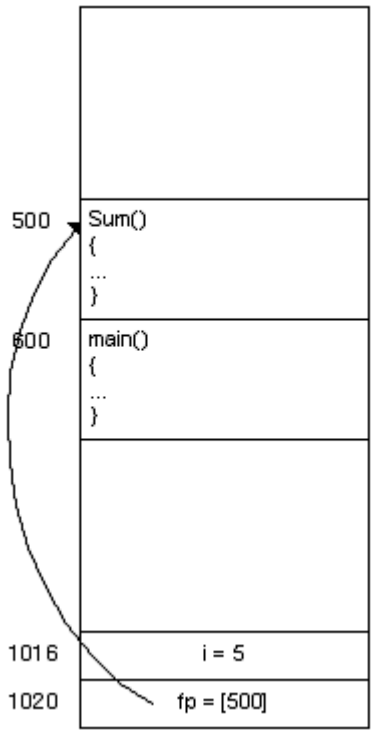
실행 결과는 다음과 같다.

```
5
```

독자들이 위 소스에서 이해해야 할 곳은 다음의 두 문장이다.

```
    fp = Sum;
    i = fp(2,3); // fp(2,3) == Sum(2,3)
```

실행 파일이 메모리에 로드(load)되면 함수나 변수들이 고유한 메모리 주소를 가지게 된다. 이렇게 함수나 변수의 실제 주소를 결정하는 것을 **바인딩(binding)**이라 한다. 바인딩된 메모리의 상태를 나타내는 아래의 [그림 2.2]를 보자.



[그림 2.2] 프로그램의 메모리 상태: Sum()함수의 시작 주소는 [500]이며 main()함수의 시작 주소는 [600]이라 가정하자. 변수 fp와 i는 각각 [1020]번지와 [1016]번지에 할당되었다. Sum()이 아닌 Sum 표현이 함수의 시작 주소 [500]을 의미한다.

Sum()함수의 시작 주소가 [500]번지에 바인딩 되었다고 가정하자. 이 때 아래의 표현식(expression)□은 Sum()함수의 시작 주소인 [500]을 의미한다.

<여기서 잠깐>
문장(statement)이 정수, 혹은 실수 처럼 어떤 값(value)을 가질 때 이러한 문장(statement)을 표현식(expression)이라 한다. 문장은 ;으로 끝나므로 쉽게 구분가능하다. 문장에서 ;을 제외한 부분이 표현식이 되려면, 그것이 값이어야 한다. 예를 들면, 2+3;이라는 문장에서 2+3은 5라는 값을 가지므로 표현식이다. 하지만, return;이라는 문장에서 return은 값을 가지지 못하므로 표현식이 아니다. 표현식은 등호(=)의 오른쪽에 사용될 수 있음을 의미한다. 함수의 이름이 표현식임을 아는 것은 중요하다. 왜냐하면 그것은 함수의 이름이 등호의 오른쪽에 사용될 수 있음을 아는 것이기 때문이다.
</여기서 잠깐>

```
Sum;
```

그리고 [500]번지에 있는 함수 즉 Sum()함수를 호출하기 위해서는 함수 호출 연산자 ()를 사용해야 한다. 즉 아래의 문장은 [500]번지의 함수를 호출하고 파라미터로 2와 3을 전달하는 것이다.

```
Sum(2,3);
```

함수의 시작 주소를 다른 변수에 저장할 필요가 있을 때, **함수 포인터 변수(pointer to a function)**를 선언한다. 이것이 변수 - 함수가 아닌 - 임에 주의하라. 함수 포인터를 선언할 때는 대입하고자 하는 함수의 원형(prototype)을 고려한 특별한 문법을 따라야 한다.

```
int (*fp)(int,int);
```

특별히 변수 fp를 둘러싼 괄호를 반드시 명시해야 한다. 괄호가 없다면 이것은 변수 선언이 아니라 함수 선언이다.

또한 int Sum(int,int)의 원형에 해당하는 부분을 명시해 주어야 한다. 사실 함수 포인터 변수를 선언하는 규칙을 외우는 것은 쉽다. Sum()의 경우 함수 선언을 위해서 아래처럼 적을 수 있다.

```
int Sum(int,int);
```

함수 포인터 변수를 선언하기 위해서는 단지 함수 이름 Sum의 앞에 *를 붙이고 괄호로 묶어주는 것밖에 없다. 아래의 소스는 int 타입을 리턴하고 int 타입을 두개 파라미터로 받는 함수 포인터 변수를 선언한 것이다.

```
int (*Sum)(int,int);
```

fp = Sum; 이라는 문장에 의해 fp도 Sum()함수의 시작 주소를 가지므로, fp()역시 [500]번지의 함수 즉 Sum()을 호출하게 되는 것이다.

함수 포인터를 이해하는 것은 매우 중요하다. 이것이 이해되지 않는다면 C++의 새로운 연산자 .*와 ->*를 이해할 수 없다.

.*, ->* 연산자

<절도비라>

MFC 메시지 맵(message map)의 핵심이 될 내용인 멤버 함수 포인터의 원리를 살펴본다. 나아가 멤버 함수 포인터의 문법과 용도를 살펴보고, 멤버 함수 포인터를 이용해서 함수를 호출하기 위한 연산자인 .*와 ->* 연산자를 이해한다.

</절도비라>

C++에서 일반 함수와 멤버 함수(member function)는 다르다. 그것은 멤버 함수가 this라는 묵시적인(implicit) 파라미터를 받기 때문이다. 그렇기 때문에 멤버 함수의 주소를 얻는 방법은 C의 방법과 다르다. .*와 ->*연산자는 멤버 함수의 주소를 얻었을 때, 특정한 객체와 연관지어 그 함수를 호출하고자 할 때 사용하는 연산자이다. 이 연산자의 역할과 문법을 확실히 알아두도록 하자. 왜냐하면 6장에서 MFC의 메시지 맵(message map)을 구현할 때 중요하게 사용되기 때문이다.

CStateManager를 클래스라 할 때 아래의 문장은 CStateManager의 멤버 함수 중 프로토타입(prototype)이 void CStateManager::Name(int); 인 멤버 함수의 시작 주소를 가지는 **멤버 함수에 대한 포인터 변수(pointer to a member function)**의 선언이다.

```
void (CStateManager::*fp)(int);
```

CStateManager의 아래와 같은 멤버 함수 SetState()를 고려해 보자.

```
void CStateManager::SetState(int state)
{
    m_iState = state;
}//CStateManager::SetState()
```

이제, 아래의 문장은 SetState()의 시작 주소를 fp에 대입한다.

```
    void (CStateManager::*fp)(int);
```

```
    fp = CStateManager::SetState;
```

이렇게 멤버 함수에 대한 포인터 변수가 선언되었을 때, 특정한 객체와 연관지어 fp를 사용하고자 할 때, .*(dot, asteriak) 혹은 ->*(minus, greater than, asterisk) 연산자를 사용한다. 아래의 [예제 2.3]을 보자.

[예제 2.3] .* 연산자의 사용

```
#include <stdio.h>

class CStateManager
{
private:
    /// 0: idle state
    /// 1: attack state
    /// 2: game over state
    int     m_iState;

    void State_Idle();
    void State_Attack();
    void State_GameOver();

public:
    CStateManager();
    void SetState(int state);
    void DoIt();
};//class CStateManager

void CStateManager::State_Idle()
{
    printf( "idle\n" );
}//CStateManager::State_Idle()

void CStateManager::State_Attack()
{
    printf( "attack\n" );
}//CStateManager::State_Attack()

void CStateManager::State_GameOver()
{
```

```
    printf( "game over\n" );
}//CStateManager::State_GameOver()

CStateManager::CStateManager()
{
    m_iState = 0; // idle state
}//CStateManager::CStateManager()

void CStateManager::SetState(int state)
{
    m_iState = state;
}//CStateManager::SetState()

void CStateManager::DoIt()
{
    switch ( m_iState )
    {
    case 0:
        State_Idle();
        break;
    case 1:
        State_Attack();
        break;
    case 2:
        State_GameOver();
        break;
    }//switch
}//CStateManager::DoIt()

void main()
{
    void (CStateManager::*fp)(int);

    fp = CStateManager::SetState;

    CStateManager   sman;
    //sman.SetState(1);
    (sman.*fp)(1); // (1)
    sman.DoIt();
}//main()
```

출력 결과는 다음과 같다.

attack

소스 (1)에서 아래의 두 문장은 같은 의미이다.

```
(sman.*fp)(1);
sman.SetState(1);
```

멤버 함수 포인터 fp가 sman의 멤버가 아니라는 사실에 주목하라. 그래서 sman.fp 혹은 sman->fp 같은 사용은 불법이다. 만약 sman이 CStateManager의 포인터 타입으로 선언된다면 (sman->*fp)(1)처럼 사용해야 한다.

.*의 사용은 클래스 멤버 변수 안에서도 코드를 자동화하는 데 사용할 수 있다. CStateManager가 게임의 상태를 유지하며, 그것이 아래의 세 가지라 가정하자.

0: idle
1: attack
2: game over

각각의 상태에 대해 DoIt()에서 호출해야 하는 함수는 아래와 같다.

```
void State_Idle();
void State_Attack();
void State_GameOver();
```

그래서 DoIt()은 다음과 같이 구현되어 있다.

```
void CStateManager::DoIt()
{
    switch ( m_iState )
    {
    case 0:
        State_Idle();
        break;
    case 1:
        State_Attack();
        break;
    case 2:
        State_GameOver();
```

```
        break;
    }//switch
}//CStateManager::DoIt()
```

이제 m_iState가 0보다 작거나 2보다 큰 값을 가지지 않는다고 가정하면 DoIt()함수를 다음과 같이 수정할 수 있다.

```
void CStateManager::DoIt()
{
    static void (CStateManager::*fp[3])() =
    {
        CStateManager::State_Idle,
        CStateManager::State_Attack,
        CStateManager::State_GameOver
    };//fp[]

    (this->*fp[m_iState])();
}//CStateManager::DoIt()
```

DoIt()의 내부에서 상태의 수에 상관없이 각 상태에 필요한 함수를 상태의 비교 없이 즉시 호출함에 주목하라. 또한, 각 상태가 독립적인 함수로 작성됨으로써, 읽기 좋고, 유지/보수가 쉬운 코드가 된다는 점도 주목하라.

.*연산자와 ->*연산자의 용도를 이해하는 것은 매우 중요하다. 우리는 6장에서 윈도우 프로시저를 위와 같은 기법으로 자동화할 것인데, MFC에서는 그것을 메시지 맵(message map)이라고 부른다.

#와

<절도비라>

이 절에서는 MFC의 코드 자동 생성기가 사용하는 매크로(macro)에서 사용되는 연산자인 스트링화 연산자 #(샵, sharp)과 토큰 연결 연산자 ##(더블샵, double shop)의 역할과 용도를 익힌다.

</절도비라>

#　　스트링(string)화 연산자
##　　토큰 연결(token concatenation) 연산자

#는 파운드 기호(pound sign)라고 읽고, ##는 더블 파운드 기호(double pound signs)라고 읽는다.

소스코드에 대한 **처리(processing)**란, 컴파일러가 기계어 코드를 생성하는 과정 - 컴파일 - 을 의미한다. 그래서 컴파일 전에 사용되는 명령문을 **전처리 명령문(preprocessing command)** 혹은 **컴파일러 지시자(compiler directive)**라 한다. 컴파일하기 전에 어떤 일을 지시하는 것이다. 위의 두 가지 연산자는 전처리 명령문에 사용되기 때문에, 전처리 연산자(preprocessing operator)로 구분된다.

#은 큰따옴표(")가 없는 문자 순서(string sequence)를 문자열로 만든다. 예를 들면 #hello_world가 #define에 사용되었다면 "hello_world"로 전처리된다. #hello world는 "hello" world로 전처리되는 것을 주의하자. 매크로 함수가 받는 파라미터를 문자열로 만드는 stringit() 매크로 함수를 아래와 같이 정의할 수 있다.

```
#define stringit(x) #x
```

위에서 처럼 stringit()이 정의되었을 때, 프로그램 소스에서 아래의 문장은 컴파일 전에 "Seo JinTaek"라고 치환된다.

```
stringit(Seo JinTaek)
```

##는 두 개의 토큰(token)■을 컴파일 전에 연결하는 연산자이다.

<여기서 잠깐>

컴파일러가 기계어 코드를 만들기 위해 처리하는 기본 단위를 토큰이라 한다. void main ( )에서 토큰은 4개이다.

</여기서 잠깐>

tokencat()이 아래와 같이 정의되었다고 가정하자.

```
#define tokencat(x,y) x##y
```

그러면 아래의 문장은 컴파일 전에 ij 로 치환된다.

tokencat(i,j)

#과 ##의 사용 예를 보여주는 아래의 소스를 참고하라.

```
#include <iostream.h>

//#define charit(x) #@x//이 연산자 #@는 각자가 사용하는 컴파일러의 도움말
                        //을 참고하라.(궁금하다면)
#define stringit(x) #x
#define tokencat(x,y) x##y

void main(void)
{
    int i=1,j=2,ij=3;

    cout << stringit(hello) << '\n';
    cout << tokencat(i,j) << '\n';
}
```

실행 결과는 다음과 같다.

```
hello
3
```

확실하게 #과 ##연산자의 역할을 이해하도록 하자. 이 연산자들은 후에 MFC의 자동 생성코드를 위한 매크로(macro)들에 광범위하게 사용된다.

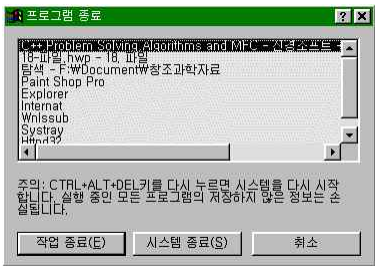
핸들(handle)

<절도비라>

운영체제가 리소스(resource) 등의 객체를 식별하기 위해서 사용하는 것을 핸들(handle)이라고 한다. 이 절에서는 바로 이 핸들의 개념을 이해한다. 또한 파일 핸들에 관한 예를 통해 파일 조작을 필요로 하는 모든 함수들이 파일 핸들을 파라미터로 받으며, 이러한 구조가 래퍼 클래스 구현의 단서가 됨을 살펴본다. 핸들에 대한 래퍼 클래스는 8장에서 구현해 본다.

</절도비라>

핸들(handle)이란 운영체제(operating system)가 특정한 정보를 유지하기 위해서, 메모리에 유지하는 정보 블록(information block)에 붙은 고유 번호(unique number)를 말한다. 예를 들면, **프로세스 컨트롤 블록(PCBs: Process Control Blocks)**은 실행 중인 각각의 프로세스의 정보를 관리한다. 프로세스가 여러 개라면, 이러한 블록은 메모리에 여러 개 유지되고 있을 것이다. 이러한 각각의 블록은 고유한 번호가 할당되는데 이 번호를 **프로세스 핸들(process handle)**이라 한다([그림 2.3]). 사용자가 프로세스에 관한 정보를 알려면, 운영체제의 함수를 호출할 때, 첫 번째 파라미터로 프로세스 핸들을 넘겨주어야 할 것이다.



[그림 2.3] Win98의 프로세스 상태: Ctrl+Alt+Del을 누르면, 프로세스 컨트롤 블락에서 정보를 읽어 태스크 윈도우(task window)를 보여 준다.

독자들은 왜 이 번호를 핸들(handle)이라 하는지 이해할 것이다. 핸들 번호만 알면, (자동차 핸들로 자동차를 조작하듯이) 이 핸들이 가리키는 정보를 마음대로 조작(handle)할 수 있는 것이다.

사용자가 파일에 관해서 입출력 작업을 하기 위해서는 먼저 **파일 핸들(file handle)**을 얻어야 한다. 파일 핸들은 **파일 제어 블록(FCBs: File Control Blocks)**에 붙여진 고유 번호이다. 파일 제어 블록은 디스크에 존재하는 파일에 입출력 작업을 하기 위해서 다양한 정보를 유지하고 있는 블록이다.

파일 제어 블록을 메모리에 할당한 다음, 디스크 파일에 관한 정보로 이 구조체 블록의 필드를 초기화하는 것을 **파일을 연다(open a file)**고 한다. 예를 들면, 파일을 열기 위해서 아래처럼 open()함수를 사용할 수 있다.

```
open("파일이름","파일모드");
```

물론 open() 함수의 파라미터로 열고자 하는 파일 이름, 파일 입출력 모드

(file I/O mode)를 설정해 주어야 한다.

파일에 관한 함수는 크게 다음과 같이 구분할 수 있다. 함수의 이름은 컴파일러에 따라 차이가 날 수 있지만, 비슷하다.

① 파일 열기(open),닫기(close) 함수
② 파일 입출력(read/write) 함수
③ 파일 포인터(file pointer)■ 조작 함수

<여기서 잠깐>
파일의 현재 읽고 쓰는 위치를 가리키는 정수타입 변수를 **파일 포인터(file pointer)**라고 한다.
<여기서 잠깐>

④ 기타(etc.) 함수

이러한 종류의 함수들은 첫번째 파라미터로 반드시 파일 핸들을 요구한다. 핸들의 개념은 빈번하게 사용된다. 또한 파라미터로 반드시 요구되는 핸들에 대한 이해는 핸들을 감싼(wrap) **래퍼 클래스(wrapper class)** 구현의 단서가 된다.

우리는 후에 8장에서 DC를 감싼 CDC, CClientDC등을 살펴 볼 것이다. 또한 파일에 대한 래퍼 클래스를 13장에서 살펴볼 것이다.

변환 연산자 오버로딩(conversion operator overloading)

<절도비라>
이 절에서는 연산자 오버로딩의 특별한 형태인 변환 연산자 오버로딩의 문법을 이해하고, 이러한 기법을 Win32 API 함수와 MFC 클래스 멤버 함수의 유연한 연결을 위해서 사용할 수 있음을 살펴본다.
</절도비라>

변환 연산자 오버로딩은 연산자 오버로딩(operator overloading)의 특별한

형태이다. 변환 연산자 오버로딩은 핸들의 래퍼 클래스에 의해 포장된 클래스와 포장되기 전의 이전 함수와의 부드러운 연결을 위해 사용할 수 있다.

핸들에 대한 래퍼 클래스 타입의 객체가 핸들을 파라미터로 받는 일반함수에 전달되는 원리를 이해하기 위해 소스를 작성해 보자. 먼저 HANDLE 타입에 대한 래퍼 클래스 CHandle을 작성한다. 이 클래스는 자신의 멤버 변수인 핸들을 리턴하는 GetHandle()을 가진다. 아래 [예제 2.4]를 보자..

[예제 2.4] GetHandle()의 사용

```
#include <stdio.h>

typedef unsigned int HANDLE;

class CHandle
{
private:
    HANDLE  m_handle;
    int     m_data;

public:
    CHandle()
    {
        m_handle = 0;
        m_data   = 1;
    }
    HANDLE GetHandle() const
    {
        printf( "GetHandle()\n" );
        return m_handle;
    }
//    operator HANDLE() const
//    {
//        printf( "GetHandle()\n" );
//        return m_handle;
//    }
    void FromHandle(HANDLE handle)
    {
        m_handle = handle;
        // construct etc. members from handle
    }
};//class CHandleMap
```

```
void Test(HANDLE handle)
{
    printf( "handle = %i\n", handle );
}//Test()

void main()
{
    CHandle     handleObject;
    HANDLE      handle;

    handle = 2;
    handleObject.FromHandle(handle);
    Test( handleObject.GetHandle() );
    //Test( handleObject );
}//main()
```

실행 결과는 다음과 같다.

```
GetHandle()
handle = 2
```

CHandle 클래스의 목적은 HANDLE을 감싸는(wrap) 것이다. 이제 CHandle 타입의 객체를 파라미터로 받는 함수가 아닌 이전의 HANDLE을 파라미터로 받는 Test()함수를 호출하는 경우에는, CHandle 객체에서 HANDLE을 얻어내는 함수 GetHandle()을 먼저 호출해야 한다.

하지만, 변환 연산자 오버로딩을 이용하면 이러한 명시적 호출을 피할 수 있고, 코드의 호환성을 높일 수 있다. 우리는 단지 아래의 예처럼 호출하기를 원한다.

```
Test( handleObject )
```

이 호출은 handleObject의 멤버인 m_handle을 Test()함수에 전달해야 한다. 우리는 이러한 경우를 위해 CHandle 타입의 객체가 HANDLE로 타입 변환(type conversion) 되어야 하는 경우 호출되는 특별한 함수를 제작할 수 있다. 아래의 [예제 2.5]를 보자.

[예제 2.5] operator HANDLE()의 사용

```
#include <stdio.h>

typedef unsigned int HANDLE;

class CHandle
{
private:
    HANDLE  m_handle;
    int     m_data;

public:
    CHandle()
    {
        m_handle = 0;
        m_data   = 1;
    }
//    HANDLE GetHandle() const
//    {
//        printf( "GetHandle()\n" );
//        return m_handle;
//    }
    operator HANDLE() const
    // 변환연산자 함수는 리턴타입이 operator
    // 뒤에 명시됨을 주의하자. 리턴타입 자리는 반드시 비워둔다.
    {
        printf( "operator HANDLE()\n" );
        return m_handle;
    }
    void FromHandle(HANDLE handle)
    {
        m_handle = handle;
        // construct etc. members from handle
    }
};//class CHandleMap

void Test(HANDLE handle)
{
    printf( "handle = %i\n", handle );
}//Test()

void main()
{
    CHandle     handleObject;
```

```
    HANDLE      handle;

    handle = 2;
    handleObject.FromHandle(handle);
    //Test( handleObject.GetHandle() );
    Test( handleObject );
}//main()
```

실행 결과는 아래와 같다.

```
operator HANDLE();
handle = 2
```

멤버 함수 operator HANDLE()은 객체가 HANDLE로 타입 변환 되는 경우 호출되는 함수를 의미한다. 이제 main()에서 아래 예의 호출은 명시적인 타입 변환 문장이 없더라도 Test()의 정의에 의해 handleObject가 HANDLE로 타입 변환되어야 함을 지시한다.

```
Test( handleObject );
```

그래서 Test()를 호출하기 전에 먼저 operator HANDLE()이 호출된다. 물론 우리는 다음과 같이 명시적 호출 버전을 작성할 수 있다.

```
Test( handleObject.operator HANDLE() );
```

변환 연산자의 역할과 용도를 이해하는 것은 MFC의 코드를 이해하는 데 필요하다. MFC 클래스들 중 핸들을 감싼(wrap) 클래스의 대부분은 원래 핸들 타입의 변환 연산자를 제공하고, 이것은 MFC 코드와 Win32 코드가 잘 호환되는 이유이다. 예를 들면 HDC를 감싼 CDC 클래스는 아래처럼 TextOut()을 멤버 함수로 제공한다.

```
TextOut(int x, int y, LPCTSTR lpszString, int nCount);
```

이제 CDC dc;처럼 선언된 CDC의 객체 dc가 있을 경우, Win32 함수

```
TextOut(HDC hdc, int x, int y, LPCTSTR lpszString, int nCount);
```

을 호출하기 위해, 아래의 예처럼 사용할 수 있다.

```
::TextOut(dc, x, y, "Hello", 5);
```

즉 ::TextOut()의 첫번째 파라미터를 결정하기 위해 dc.operator HDC()가 호출되는 것이다.

<여기서잠깐>
TextOut()의 앞에 범위 해결사(scope resolver)를 적어주는 것은 중요하다. 이것은 문맥이 모호한 상황에서 CDC의 멤버 함수 TextOut()이 아니라, API 함수 TextOut()을 호출하는 것을 명확히 한다.
</여기서잠깐>

요약

이 장에서 MFC 소스 분석에 필요한 C++의 주제들 중 첫 부분을 살펴보았다. 각 주제들은 MFC의 소스 분석과정에서 곳곳에 사용될 것이다.

윈도우 프로그래밍을 시작하기 전에 C++의 모든 것에 대한 자세한 이해가 필요한 것은 아니지만, 윈도우 프로그래밍 연습을 마칠 때는 C++의 모든 주제들을 자세하게 이해하고 있어야 한다.

- 상태의 on/off를 나타내는 변수를 **플래그(flag)**라 한다. 여러 상태의 on/off를 나타내기 위해 정수의 각 비트들을 사용할 수 있는데, 이것을 **비트 플래그(bit flag)**라 한다. 비트 플래그는 API 함수가 많은 상태를 전달하거나 받기 위해 사용하는 일반적인 방법이다.
- **함수 포인터**란 함수의 시작 주소를 가리키는 특별한 포인터 변수이다. 함수 포인터 변수를 이용해 함수를 간접적으로 호출할 수 있으며, 이것은 일반적이고 대중적인 방법이다.
- **.*와 ->* 연산자**는 특별하게 선언된 (특정 객체의 멤버가 아닌) **멤버 함수에 대한 일반 포인터 변수**를 특정 객체가 접근하기 위한 연산자이다. .*와 ->* 연산자의 앞에는 객체가 있어야 하지만, .*와 ->* 연산자 뒤에는 객체의 멤버가 아닌 멤버 함수에 대한 일반 포인터 변수가 와야 한다. .*와 ->* 연산자는 6장에서 메시지 맵(message map)을 구현하기 위해 중요하게 사용한다.

- **#과 ##연산자**는 전처리 명령문에서만 사용되는 특별한 연산자이다. #은 매크로 함수에 전달된 파라미터를 문자열로 만들기 위해 사용하고, ##은 매크로 함수에 전달된 토큰을 연결한 새로운 토큰을 만들기 위해 사용한다.
- **핸들(handle)**은 구조체를 구분하는 유일한 ID로써, 정수이거나 포인터 자체가 될 수도 있다.
- **변환 연산자 오버로딩**은 연산자 오버로딩의 특별한 종류로 타입 변환 시에 호출되는 연산자 함수이다. 변환 연산자 오버로딩을 잘 사용하면 기존에 존재하는 함수와 래퍼 객체가 호환되는 코드를 작성할 수 있다.

[문서의 끝]