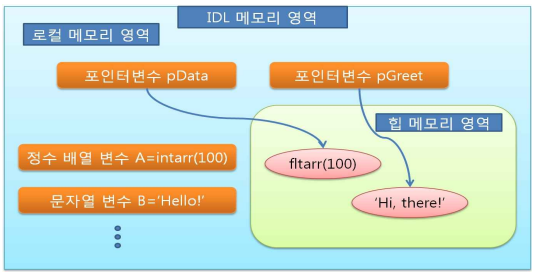
IDL에도 포인터가 있습니다.

IDL의 포인터와 C 언어의 포인터

포인터는 C언어의 상징과 같은 데이터 타입입니다. 실제 메모리를 마구 뛰어다닐 수 있는 포인터의 사용은 C언어를 다른 어떤 언어보다도 강력하게 만들지만, 처음에는 이해하기 까다롭게 보이기도 합니다.
IDL에도 포인터 타입이 존재합니다. C언어의 포인터와 다른 점은 다음의 두 가지 정도로 볼 수 있겠습니다.
1) C의 포인터가 실제 메모리를 가리키는 주소인 반면, IDL의 포인터는 IDL이 제공하는 힙 메모리(Heap Memory) 공간 안에서 특정 데이터로 접근하는 주소입니다. 그러므로 C의 포인터를 잘못 쓰면 시스템 운영에 심각한 손상을 줄 수도 있지만, ***IDL의 포인터는 IDL 영역 밖의 문제를 만들지 않습니다.***.
2) C언어는 포인터 없이 제대로 된 프로그램을 만들 수 없지만, ***IDL에서는 대부분의 프로그래머가 포인터 없이 잘 지내고 있습니다.*** 그렇기는 해도, IDL에서 포인터를 잘 다룰 수 있다면 프로그램이 훨씬 효율적이 되는 경우가 있어 고급 사용자가 되기 위해서는 꼭 배워야 합니다.

힙 메모리와 포인터

IDL에서 일반적으로 사용하는 메모리 공간은 로컬 메모리라고 하며, 함수나 프로시저가 실행될 때마다 필요한 영역이 생성되었다가 종료될 때 함께 소멸합니다. 이러한 ***로컬 메모리 공간은 전적으로 IDL이 관리하며, 메모리를 보존할 방법도, 기존의 메모리 영역을 재사용할 방법도 존재하지 않습니다.***
이에 반해, ***힙 메모리는 전적으로 사용자가 관리합니다.*** 힙 메모리 공간은 IDL이 제공하지만, 이 안에 데이터를 쓰고 지우는 것은 사용자의



힙 메모리와 포인터

재량입니다. IDL이 종료되는 경우를 제외하고는 힙 영역은 자동으로 지워지지 않습니다.
힙 메모리 공간에는 메모리가 허용하는 한, 어떤 데이터 타입이든 몇 개든 마음대로 데이터를 쓸 수 있습니다. 그렇다면, 힙 메모리 공간에 저장된 여러 가지 데이터 중에서 우리가 필요로 하는 데이터를 찾아낼 방법이 있어야 할텐데요. ***힙 메모리 공간의 특정 데이터를 가리키고 있는 화살표와 같은 데이터 타입이 바로 포인터입니다.*** 이런 이유로 포인터(Pointer)라는 이름을 가지게 된 것입니다. 힙 메모리에 저장한 데이터는 포인터를 통해서만 사용할 수 있게 됩니다.

IDL 포인터 기초

포인터 생성 : PTR_NEW()

```
IDL> pa=ptr_new(18.44)
```

어떤 종류의 데이터든지 힙에 넣고 포인터를 만들 수 있습니다. 위 예는 Float형 18.44를 힙에 넣고 이를 가리키는 포인터 변수 pa를 생성한 것입니다. 관례적으로 포인터 변수 이름을 p로 시작하지만 꼭 그래야 하는 것은 아닙니다.

포인터의 데이터 타입 : Pointer 형입니다.

```
IDL> help, pa
PA      POINTER = <PtrHeapVar1>
```

'힙 영역의 포인터 1번'을 의미합니다.

포인터가 가리키는 힙의 실제 데이터 : *을 이용합니다.

```
IDL> help, *pa
<PtrHeapVar1> Float = 18.4400
```

포인터 pa가 가리키는 힙 영역에 저장된 값은 18.44입니다. 이처럼 포인터가 가리키는 곳의 값을 참조하는 것을 dereferencing이라고 합니다.

여러 포인터가 하나의 힙 영역을 가리킬 수 있습니다.

```
IDL> pb=pa
IDL> help, pb
PB      POINTER = <PtrHeapVar1>
```

PB는 PA와 같은 곳을 가리키는 포인터입니다.

```
IDL> *pb=3.14
IDL> print, *pa
   3.14000
```

포인터가 가리키는 힙 메모리 삭제 : PTR_FREE

```
IDL> ptr_free, pa
IDL> help, pa
PA      POINTER = <PtrHeapVar1>
```

여러 포인터가 하나의 힙 영역을 가리킴

포인터는 그대로 남아 있지만, 실제 힙 메모리의 데이터는 소멸되었습니다. 포인터 PA는 로컬 변수이므로 프로그램이 종료되면 자동으로 삭제됩니다. 힙 영역 데이터는 사용하지 않게 되면 반드시 위와 같이 사용자가 직접 제거해 주어야 합니다.

포인터가 가리키는 힙 데이터의 유효성 검사

```
IDL> print, ptr_valid(pa), ptr_valid(pb)
  0  0
```

pa, pb 모두 같은 곳을 가리켰으나 힙의 내용을 제거하였으므로 유효하지 않음(0)을 리턴합니다.

배열의 포인터

```
IDL> arr=[92.3, 17.2, 44.5, 72.9]
IDL> parr=ptr_new(arr)
IDL> print, (*parr)[1:3]
 17.200    44.500    72.900
```

*parr 이 배열입니다. 그러므로 (*parr)[0] 과 같이 *parr을 먼저 계산하도록 괄호로 싸서 배열을 힙으로부터 꺼내온 다음 배열 번호로 참조해야 합니다.

구조체의 포인터

```
IDL> rec={name:'Mercury', a:0.387, e:0.206}
IDL> prec=ptr_new(rec)
IDL> print, (*prec).name, (*prec).e
Mercury    0.206
```

NO_COPY 키워드

```
IDL> a=findgen(10)
```

```
IDL> pa=ptr_new(a)
IDL> help, a
A       Float = Array[10]
IDL> pb=ptr_new(a, /NO_COPY)
IDL> help, a
A       UNDEFINED = <Undefined>
```

로컬 변수를 힙 영역에 저장하는 경우 로컬의 내용을 그대로 두고 힙에 복사를 하게 됩니다. /NO_COPY 키워드를 사용하면 힙으로 데이터를 이동하고 로컬 메모리의 내용은 소멸됩니다.

유효한 포인터 목록

PTR_VALID() 함수를 인자 없이 사용하면, 힙 메모리를 가리키는 유효한 포인터 모두를 리턴합니다.

```
IDL> print, ptr_valid()
<PtrHeapVar2><PtrHeapVar3><PtrHeapVar4><PtrHeapVar5>
```

그러므로 다음과 같이 모든 유효 포인터가 가리키는 힙 메모리를 소멸시킬 수 있습니다.

```
IDL> ptr_free, ptr_valid()
```

포인터, 어디에 써먹으라는 겁니까?

일반적인 IDL 사용자들이 포인터를 쓸 일이 많지는 않을 것입니다. IDL 사용자들이 포인터를 주로 사용하는 분야는 다음과 같습니다.

포인터의 배열

배열은 그 안의 모든 요소들이 같은 데이터 타입을 가져야 합니다. 예를 들어, 정수와 실수가 같은 배열에 들어갈 수 없습니다.

```
IDL> arr=[0B, 11.3, 14.7d]
IDL> print, arr
 0.0000000    11.3000000    14.7000000
```

Byte형, Float형, Double형을 하나의 배열에 넣을 수 있는 것처럼 보이지만 사실은 모두 최상위 타입인 Double 형으로 변환되어 배열이 됩니다. 다음과 같은 경우는 아예 되지도 않습니다.

```
IDL> arr=[0, 'Funny']
%Type conversion error: Unable to convert given STRING to Integer.
```

포인터는 모두 같은 타입입니다. 포인터가 가리키는 힙의 내용은 IDL의 어떤 종류 데이터라도 가능합니다. 그러므로 다양한 종류의 데이터를 모두 힙에 넣고, 이들을 가리키는 포인터만 배열로 만들면 위의 시도를 성공시킬 수 있습니다.

```
IDL> ptry=ptrarr(2)
IDL> ptry[0]=ptr_new(0)
IDL> ptry[1]=ptr_new('Funny')
IDL> for I=0,1 do print, *ptry[i]
   0
Funny
```

구조체 내의 포인터

구조체가 한번 정의되고 나면, 필드의 타입 및 크기를 변경할 수가 없습니다.

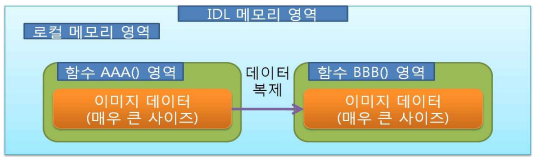
```
IDL> person={id:0, picture:bytarr(50,50)}
IDL> person.picture=intarr(100,100)
% Conflicting data structures .....
```

향후 타입과 크기가 변경될 수 있는 필드를 포인터 타입으로 만들어 놓으면 포인터가 가리키는 곳의 데이터는 어떤 타입, 어떤 크기든 넣을 수 있게 됩니다.

```
IDL> person2={id:0, picture:ptr_new(bytarr(50,50))}
IDL> *person2.picture=intarr(100,100)
IDL> help, *person2.picture
<PtrHeapVar2>    INT   = ARRAY[100,100]
```

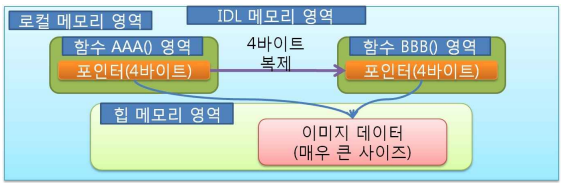
대용량 자료의 공유

그림과 같이 AAA라는 함수가 BBB라는 함수에 큰 이미지 데이터를 넘겨주는 경우를 생각해 보면, AAA 함수의 메모리 공간에도 이미지 데이터가 존재하고, BBB함수의 메모리 공간에도 이미지 데이터를 복사해 넣게 됩니다. 두 개의 동일한 데이터가 생성되는 것도 비효율적이라 볼 수 있고, 새로 메모리 공간을 생성하는 데에도 시간이 많이 걸릴 수 있습니다.



두 모듈 간의 데이터 직접 전송

포인터를 사용하는 경우라면, 힙 공간에 큰 이미지 데이터를 써넣고 이를 포인터로 가지고 있다가 다른 함수에 데이터를 넘겨 주어야 할 때에는 포인터만 넘겨 주면, 같은 힙 공간에서 데이터를 사용할 수 있게 됩니다. 이미지의 크기가 얼마든 간에, 포인터의 크기는 4byte 밖에 되지 않습니다.



포인터를 이용한 가벼운 데이터 전송

GUI 프로그래밍은 최소 2개의 모듈이 상호간의 거의 모든 데이터를 공유하며 실행이 됩니다. 각 모듈간에 호출이 일어날 때마다 이런 데이터를 매번 복사해야 한다면 프로그램의 성능은 매우 낮아질 것입니다. 이 때문에, GUI 프로그래밍에는 포인터가 필수적으로 사용되고 있습니다.

험한 꼴, 포인터가 없어져 버린 힙 메모리

힙 메로리를 가리키는 포인터가 없어진다면, 연결되어 있던 힙 메모리는 이제 사용할 수 없게 되는 험악한 상황입니다. 디버깅의 경우라면, PTR_VALID() 함수에 /CAST 키워드를 써서 다시 살릴 수 있는 상황도 있지만, 실행 중인 프로그램에서는 이렇게 할 수가 없습니다.

```
IDL> p1=ptr_new(DIST(100))
IDL> p1='Do not be silly.'
```

이렇게 되면 p1은 더 이상 포인터가 아니며, p1이 가리키고 있던 DIST(100) 이 저장된 힙 메모리를 찾아갈 방법이 없어집니다.

```
IDL> help, /heap   ;고아가 된 힙 메모리가 보입니다.
Heap Variables:
<PtrHeapVar3> FLOAT = ARRAY[100, 100]
```

포인터가 없어 쓸모가 없어진 힙 메모리의 데이터들을 가비지(Garbage; 쓰레기)라고 하며, 이들은 괜히 메모리만 차지하고 있게 되므로 존재 자체가 바람직하지 않습니다. 이런 쓰레기들을 정리하는 일을 '가비지 콜렉션'이라고 하며 IDL에서는 다음과 같이 합니다.

```
IDL> heap_gc, /verbose                ;힙의 가비지를 정리
```

가비지 콜렉션은 시스템 부하가 커질 수도 있어 가능한 하지 않도록 미리 관리해야 합니다. 힙 메모리를 놓치는 일은 프로그래머가 해서는 안되는 일 중 하나입니다. 포인터를 잘 관리하고, 사용이 끝난 힙 메모리는 PTR_FREE로 반드시 정리하는 것이 좋은 습관입니다..