## Example : My Own Storage Management in C++

9/21

Pool Management 을 class

```
typedef struct ptr_list{
  int i;
  struct ptr_list *p;
} ptr_L;

class Linked_List {
protected:
  static ptr_L *ptr_L_pool;   // 공유 storage이므로
  static int ptr_L_cnt;       // static으로 선언
public:
  ptr_L* Alloc_ptr_L(void);
  void Free_ptr_L( ptr_L *ptr );
  void Free_ptr_L_pool ( void );
};

// 위에서 static으로 선언하였으므로 global 선언필요
ptr_L* Linked_List::ptr_L_pool = NULL;
int    Linked_List::ptr_L_cnt = 0;
```

값을 update 하면 그대로 유지.

need to delcare Used_Memory_for_ptr

## Membership Functions

```
ptr_L* Linked_List::Alloc_ptr_L(void){
  ptr_L *ptr;
  if ( ptr_L_pool == NULL ) {
    //ptr = (ptr_L *)malloc(sizeof(ptr_L));
    ptr = new ptr_L;
    if ( ptr == NULL ){
          printf("memory alloc error\n");
          exit(1);
    }
  }
  else {
    ptr         = ptr_L_pool;
    ptr_L_pool = ptr->p;
  }
  ptr->i   = 0;
  ptr->p   = NULL;
  ++ptr_L_cnt;
  return(ptr);
}
```

Update Used_Memory_for_ptr

C 한줄 설명

## Membership Functions (cont'd)

```
void Linked_List::Free_ptr_L( ptr_L *ptr ) {
  ptr->p      = ptr_L_pool;
  ptr_L_pool = ptr;
  --ptr_L_cnt;
}

void Linked_List::Free_ptr_L_pool ( void ){
  ptr_L *p;
  if ( ptr_L_cnt != 0 ){
    printf("memory free error\n");
    exit(1);
  }
  while (ptr_L_pool != NULL ) {
    p = ptr_L_pool;
    ptr_L_pool = p->p;
    //free(p);
    delete p;
  }
}
```

Update Used_Memory_for_ptr

## 구현 대상

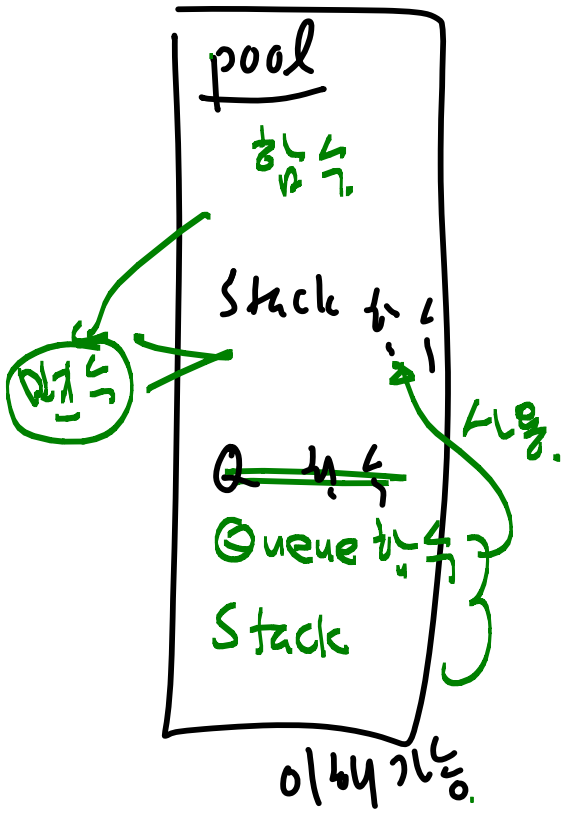
Class 고민

- Linked List 관리
  - pool
  - Alloc ptr
  - Dealloc ptr

Stack insert
delet

Queue insert
delete

고민: multiple stack, queue를 관리할 수 있어야 함 (independent하게) S1 Q1 Q2




C++

구상이 필요하다
Class를 어떻게 define!

- 강의노트의 처음에 그림을 보인다.
- 다른 구현방법을 있을 것임.

- Linked_List class
- Stack } 상속 classes
- Queue }

## ✦ Terminologies


◆**private** : 이 지정 이후의 멤버는 클래스 내의 멤버 함수만 액세스 가능. 클래스 밖에서 안보이며 클래스에 속하지 않는 함수는 이 속성의 멤버를 액세스 할 수 없다.

◆**public** : 이 지정 이후의 멤버는 클래스 외부로도 알려지며 외부 함수가 읽거나 쓸 수 있으며 멤버 함수가 이 속성을 가질 경우 외부에서 멤버 함수를 임의로 호출할 수 있다. 단, 통용 범위 규칙에 의해 지정되는 범위 내에서만 가능하다.

◆**protected** : private와 마찬가지로 동일 클래스 내부의 멤버 함수에 의해서 액세스가 가능하며 클래스 밖의 함수는 액세스 수 없다. private와 다른 점은 상속이 될 경우 상속된 클래스에 속한 멤버 함수들도 이 protected 속성의 멤버를 액세스 할 수 있다는 점이다. private의 경우 상속이 되면 상속된 클래스의 멤버 함수들은 이 멤버를 액세스 할 수 없으며 오직 해당 클래스 내의 멤버 함수만 액세스 할 수 있다. 상속하지 않는 클래스라면 private속성이나 protected 속성이나 차이가 없다.

## ✦ Terminologies(cont'd)

◆ constructor : 클래스 객체가 생성될 때 한번만 호출되는 멤버십 함수이다. 이 함수의 이름은 클래스이름과 같고 return type을 갖지 않는다. 주로, 클래스 내의 멤버 변수들을 초기화하는데 사용한다.

- new 또는 객체를 선언할 때 한번 호출된다.

◆ destructor : 클래스 객체가 해제될 때 한번만 호출되는 맴버십 함수이다. 이 함수이름은 클래스이름 앞에 ~를 붙여서 명명한다. Constructor와 마찬가지로 return type이 없다. 주로, 메모리를 해제하는데 사용한다.

- delete 사용시 호출된다.
- Local 객체일 경우 함수 종료시 호출된다. 메인에서 생성한 객체일 경우 프로그램 종료시 호출된다.

◆ 참고 : constructor와 destructor를 선언하지 않았을 경우 default 함수가 호출되나 아무런 동작을 하지 않는다.

◆ 앞의 Linked_List 객체에서는 이들을 선언하지 않았다(불필요).

## Stack Implementation in C++ (1/2)

```
class stack:public Linked_List{
private:
  ptr_L *Q_ptr;
  //Init_Stack함수 private으로 설정
  void  Init_Stack(void);
public:
  stack(void);  //constructor
  ~stack(void); //destructor
  bool  Empty_Check(void);
  void  pushQ_ptr_L(int i);
  int   popQ_ptr_L(void);
  void  Free_Stack(void);
};
void stack::Init_Stack(void){
  Q_ptr = NULL;
}
bool stack::Empty_Check(void){
  if(Q_ptr == NULL) return true;
  else              return false;
}
```

class stack은 class Linked_List의 속성을 상속 받는다.

```
//constructor
stack::stack() {
    Init_Stack();
}
//destructor
stack::~stack() {
    Free_Stack();
}
```

## Stack Implementation in C++ (2/2)

```
void stack::pushQ_ptr_L(int i){
  ptr_L *L;
  L    = Alloc_ptr_L();
  L->i = i;
  L->p = Q_ptr;
  Q_ptr   =  L;
}
int stack::popQ_ptr_L(void){
  int      i;
  ptr_L    *tQ;
  i    = Q_ptr->i;
  tQ   = Q_ptr;
  Q_ptr = Q_ptr->p;
  Free_ptr_L( tQ );
  return ( i );
}
```

```
void stack::Free_Stack(void) {
  ptr_L *p;
  while ( Q_ptr != NULL ) {
    p = Q_ptr;
    Q_ptr = Q_ptr->p;
    Free_ptr_L(p);
  }
}
```

pool

## Queue Implementation in C++(1/3)

```
class queue:public Linked_List {
private:
  ptr_L *Q_ptr, *Q_ptr_end;
  //Init_Queue함수 private로 설정
  void  Init_Queue(void);
public:
  queue(void);  //constructor
  ~queue(void); //destructor
  bool  Empty_Check(void);
  void  addQ_ptr_L (int i);
  int   getQ_ptr_L (void);
  void  Free_Queue(void);
};
void queue::Init_Queue(void){
  Q_ptr = Q_ptr_end = NULL;
}
bool queue::Empty_Check(void){
  if (Q_ptr == NULL) return true;
  else               return false;
}
```

```
//constructor
queue::queue() {
    Init_Queue();
}
//destructor
queue::~queue() {
    Free_Queue();
}
```

## Queue Implementation in C++(2/3)

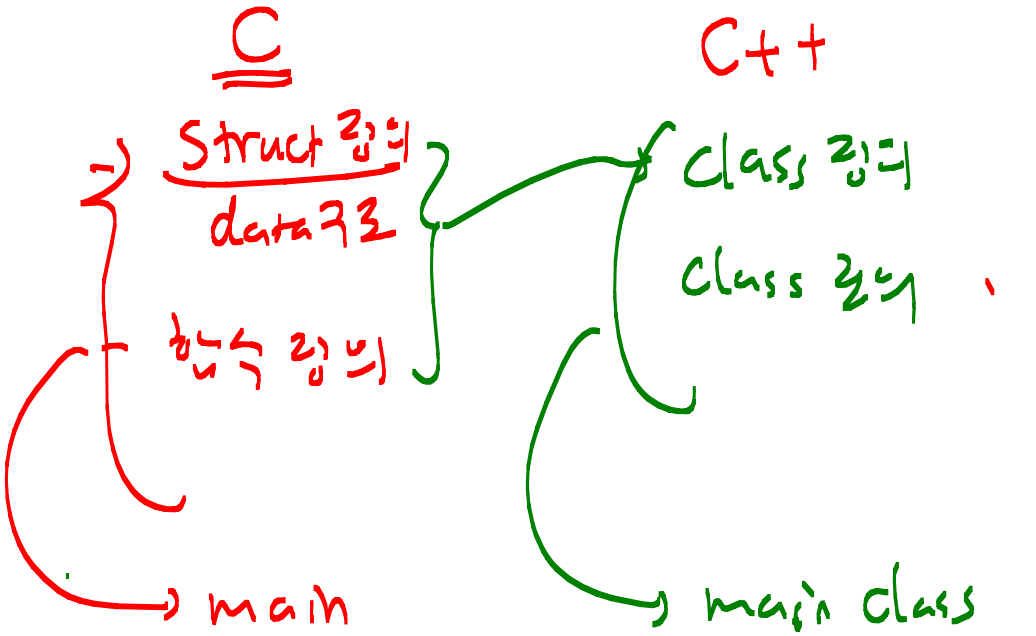
```
void queue::addQ_ptr_L(int i){
  ptr_L *L;
  L    = Alloc_ptr_L();
  L->i = i;
  if ( Q_ptr == NULL )
    Q_ptr = L;
  else
    Q_ptr_end->p = L;
  Q_ptr_end = L;
}
```

```
int queue::getQ_ptr_L(void){
  int   sym;
  ptr_L *tQ;
  sym  = Q_ptr->i;
  tQ   = Q_ptr;
  if ( Q_ptr->p == NULL ) {
    Q_ptr = Q_ptr_end = NULL;
  }
  else
    Q_ptr = Q_ptr->p;
  Free_ptr_L(tQ);
  return (sym);
}
```

# ◆ Queue Implementation in C++(3/3)

```
void queue::Free_Queue(void){
  ptr_L *p = Q_ptr;
  while ( Q_ptr != NULL ) {
    p = Q_ptr;
    Q_ptr = Q_ptr->p;
    Free_ptr_L(p);
  }
  Q_ptr_end = NULL;
}
```

C++ 구성이 필요




요즘 회사에서는? C++ 코딩인

Why C? 컴공과에서는 기초적인.

Java → class, 또는 컴파일된 class → 또는 C++로 만든다.

## An Example of Using Stacks

### main 함수인 경우

```
stack list1, list2; // create two stacks
//constructor가 각각 호출되어 초기화 한다(Init_Stack())

for(int i=0; i<10; i++) { // push 0-9 to stack1
  list1.pushQ_ptr_L(i);
}
while(list1.Empty_Check()) {      // pop from stack1
  list1.popQ_ptr_L();
}
for(int i=0;i<11;i++){ // push 0-10 to stack2
  list2.pushQ_ptr_L(i);
}
list1.Free_Stack(); // 이미 empty이므로 불필요.
list2.Free_Stack(); // clear stack2

list1.Free_ptr_L_pool(); // Clear pool (call once)
// 프로그램 종료시 list1, list2에 대한 destructor가 자동
// 호출되지만 이미 empty인 상태이므로 아무런 영향이 없다.
```

✦sub 함수인 경우

```
void Test_Stack ( void ) {
  stack list; // create stack in local function
              // constructor가 자동 호출된다.
  for(int i=0; i < 10; i++) {// push 0-9 to stack1
    list.pushQ_ptr_L(i);
  }
}
// return시 list에 대한 destructor가 자동 호출된다.
// 따라서 list.Free_Stack()을 따로 호출할 필요가 없다.
```

- STL(Standard Template Library)
  - C++에서 여러 자료 구조와 알고리즘을 구현한 라이브러리.
    - list, stack, queue, map, set 등 여러 자료구조 와 알고리즘을 제공.
    - C++언어의 template 기능을 이용하여 자료의 유형에 상관없이 저장 가능(generic). (http://ko.wikipedia.org)
  - References
    - STL Online Reference : http://www.cplusplus.com/reference/stl/
    - STL Sample Codes :

      http://msdn.microsoft.com/enus/library/f1dtts6s%28v=VS.90%29.aspx
    - Book : "STL Tutorial and Reference Guide : C++ Programming With the Standard Template Library", David R. Musser, Gillmer J. Derge, Atul Saini, Addison-Wesley.

✦STL의 사용 예 : stack

◆<stack> 헤더 파일을 포함해야 한다:

◆ `std::stack <data type> s;`

```
s.push();  //stack에 원소를 삽입
s.top() ;  //stack에서 가장 최근에 삽입된 원소를 반환
s.pop() ;  //stack에서 가장 최근에 삽입된 원소를 제거
s.size();  //현재 원소의 개수를 반환
s.empty(); //stack이 비어있는지 판단
```

◆Example

```
std::stack <int> list1,list2; // create two stacks
for(int i=0; i<10; i++) { // push 0-9 to stack1
  list1.push(i);
}
while(!list1.empty()) { // pop from stack1
  list1.pop();
}
for(int i=0;i<11;i++){ // push 0-10 to stack2
  list2.push(i);
}
```

## ✦STL의 사용 예 : queue

- ◆<queue> 헤더 파일을 포함해야 한다:  #include <queue>
- ◆ std::queue <data type> q;

```
q.push() ; //queue에 원소를 삽입
q.front(); //queue에서 가장 오래전에 삽입된 원소를 반환
q.back() ; //queue에서 가장 최근에 삽입된 원소를 반환
q.pop()  ; //queue에서 가장 오래전에 삽입된 원소를 제거
q.size() ; //현재 원소의 개수를 반환
q.empty(); //queue가 비어있는지 판단
```

## ✦STL의 사용 예 : list (doubly linked list)

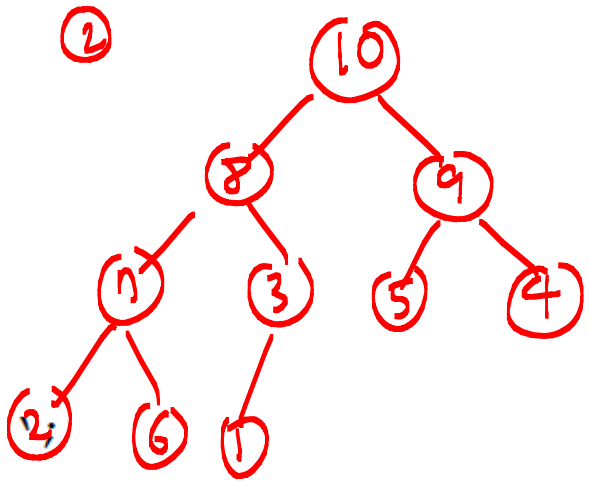
◆<list> 헤더 파일을 포함해야 한다: #include <list>

◆
```
std::list<data type> l;
l.front() ;          //list에서 가장 앞에 있는 원소를 반환
l.back() ;           //list에서 가장 뒤에 있는 원소를 반환
l.push_front(m) ;    //list의 앞에 원소 m을 추가
l.pop_front() ;      //list의 제일 앞의 원소를 제거
l.push_back(m) ;     //list의 제일 뒤에 원소 m을 추가
l.pop_back() ;       //list의 제일 뒤에 원소를 제거
l.remove(v) ;        //list에서 값이 v인 모든 원소를 제거
l.size() ;           //현재 원소의 개수를 반환
l.empty() ;          //list가 비어있는지 판단
```

◆List 값을 순차적으로 읽을 때 iterator를 사용할 수 있다.

```
std::list<int> l;
for(int i=0; i<10; ++i) // insert 0-9 into list
  l.push_front(i);
for(list<int>::iterator i=l.begin(); i!=l.end(); ++i)
  cout<<*i<<endl;  // list의 원소를 순서대로 출력
```

- STL의 사용 예 : heap
  - #include <algorithm> 필요
  - Example



```
#include <algorighm>
int main(void){
  int ar[10] = {2,3,9,7,1,5,4,8,6,10};
  for(int i=0;i<10;i++)    cout<<ar[i]<<' ';
  cout<<endl; //Print out new line (1)
  std::make_heap(ar, ar+10); //Make max heap
  for(int i=0;i<10;i++)    cout<<ar[i]<<' ';
  cout<<endl; //Print out new line (2)
  std::sort_heap(ar, ar+10); //Sort ar[] by inc order
  for(int i=0;i<10;i++)    cout<<ar[i]<<' '; (3)
}
        //Output
        //2 3 9 7 1 5 4 8 6 10 (1)
        //10 8 9 7 3 5 4 2 6 1 (2)
        //1 2 3 4 5 6 7 8 9 10 (3)
```

- algorithm.h
  - STL에서 제공하는 자료구조들의 원소를 탐색 및 변경하는 함수들을 제공.
  - find : 지정한 범위에서 원하는 원소 값을 찾는다.
  - swap: 2개의 원소를 교환한다.
  - sort : 지정한 범위의 원소들을 정렬한다.
  - binary_search : 정렬된 배열에서 지정한 원소 값을 찾는다.
  - make_heap : 주어진 배열을 max(min) heap으로 만든다.
  - min/max : 최소/최대값을 찾는다.
  - etc.

- STL의 다른 header files
  - <bitset> : 비트 또는 Boolean 값을 위한 고정 사이즈 배열.
  - <map> : 임의의 원소에 대한 접근을 로그시간으로 보장하는 정렬된 연관 컨테이너. 원소의 키를 원소의 값과 대응시켜 검색한다. 모든 삽입/삭제/탐색에 대해 로그시간을 보장.
  - <pair> : 두 개의 데이터를 하나의 object로 처리할 수 있게 해줌. 첫 번째 인자는 first라고 하고, 두 번째 인자는 second라고 하며 이 순서는 고정되어 있다.
  - <set> : 임의의 원소에 대한 접근을 로그시간으로 보장하는 정렬된 집합. 모든 삽입/삭제/탐색에 대해 로그시간을 보장.
  - <vector> : 크기 조절이 가능한 C 스타일의 동적 배열처럼 동작한다. 임의의 원소에의 접근이 상수시간으로 보장되며, 배열의 끝에서의 삽입/삭제도 상수시간이 보장된다.
  - etc…