**CS-202**

# Dynamic Memory (Pt.1)

**C. Papachristos**

**Autonomous Robots Lab**
**University of Nevada, Reno**

Course , Projects , Labs:

| Monday | Tuesday | Wednesday | Thursday | Friday |
|--------|---------|-----------|----------|--------|
| | | | Lab (4 Sections) | |
| | CLASS | RL – Session | CLASS | |
| PASS Session | PASS Session | **Project DEADLINE** | NEW Project | |

Your 6th Project Deadline was past Sunday 3/17.

➢ No PASS Sessions this Monday-Tuesday !

➢ No RL Session this Wednesday !

➢ However, a NEW Project is going to be announced on Thursday !

# Today's Topics

Memory Storage – ( Basics )
- Automatic
- Static
- Dynamic

Program Memory
- Stack
- Heap

Program Memory Management
- Expression `new` ( `[ ]` )
- Expression `delete` ( `[ ]` )

## Program Data

*Automatic* Storage Duration:

Reside in activation frame of the function, destroyed when returning from function. Automatically created at function entry.

➢ Objects declared at function Block Scope.
➢ Objects declared in function Parameter Lists.

Note:
➢ Storage specifier `auto` was sometimes used in older standards to declare such storage duration:
  e.g. `auto int myIntVal;`
Attention:
➢ From C++11 and onwards, auto is used as type deduction specifier:
  e.g. `auto myVal;`

**Program Data**

*Static* Storage Duration:

Memory allocation takes place at compile-time before the associated program is executed.

Static memory storage covers the entire program lifetime.

➢ All objects declared at namespace scope (including the global namespace).
➢ Only one instance of the object exists.

Usual examples:
➢ Global variables.
➢ `static` local variables (in functions).
➢ `static` member variables (in Classes).
➢ Virtual Function Tables (Polymorphism).

Note (Twice-Initialized case of `static` variables):
➢ Allocated at program start & "early 0-initialized", (i.e. before any other initialization takes place.
➢ Initialized (actual value-based initialization or constructor call) by program the fist time they are encountered in translation unit.

## Program Data

*Dynamic* Storage Duration:

Explicit programmer-made allocation / deallocation calls in C++.
Allocated and deallocated at run-time per-request.

➢ Usage of C++ Memory Management Functions.

Usual examples:
➢ Operator `new` .
➢ Operator `delete` .

Low-level C++ Memory Management

Note: Well-known C-style memory management functions are now part of the `<cstdlib>` header:
➢  `void* std::malloc(std::size_t)` , `void* std::calloc(std::size_t,std::size_t)`
➢  `void std::free(void*)` , `void* std::realloc(void*, std::size_t)`

**Program Data**

*Thread–Local* Storage Duration (not to our interest now):

Allocated when a thread begins and deallocated when that thread ends.

➢ Each thread has its own instance of the object.

More information and complete reference:

➢ http://en.cppreference.com/w/cpp/language/storage_duration

Complete reference on:

➢ Storage Specifiers & Storage Duration.
➢ Storage Specifiers & Linkage.

**Memory Allocation**

*Static* Allocation:

Management of memory (De)-Allocation predescribed at compile-time.
➢ (*Remember*: *Static* binding – takes place at compile-time).

*Dynamic* Allocation:

Management of memory (De)-Allocation performed at run-time.
➢ (*Remember*: *Dynamic* binding – takes place at run-time).

## Static Allocation

*Static* Allocation is handled automatically (implicitly) at compile-time.

➢ Global variables or objects:
   Memory allocated (actually loaded) at the start of the program, and free'd when program exits; alive throughout program.
   > Note: Actually, these live in the Initialized/Uninitialized Data (`.data`) Segments.
      Direct address-based accessing can be guaranteed to succeed from anywhere in the program.

➢ Local variables (inside a function Block Scope – the `main()` included):
   Memory allocated when function starts and free'd when the function `return`s.
      Local variables cannot be accessed from outside the function Block Scope.

**Static Allocation**

*Static* Allocation is handled automatically (implicitly) at compile-time.

➢ No need to provide explicit handling for memory management.
➢ Easy to work with, but with certain limitations.

With *Static* Allocation, all variable / object storage must also be known at compile-time.

➢ Necessary to have known-composition data types (Class/Struct members, etc.).
➢ Necessary to have fixed-size arrays (of some `MAX_SIZE` dimensions).
➢ Simple counter-point against it: What about a container that needs to grow (and potentially shrink) to the program's needs ?

## The Stack

The Stack is part of the Program Virtual Memory. It is used to hold all necessary information about the active functions (at run-time).

➢ This includes its Local Variables (as well as parameters, return address, etc.)
➢ All Local Variables will take up space from the Stack.

Do(s):
➢ Very fast memory allocation (but strict size limitations – Thread creation / O.S. limitations / etc.)
➢ Use for short-lived and "small" data:

```
if (condition){
    double dArr[5];
    /* dArr manipulations */
}  /* dArr out of scope, free'd */
```

Auto storage duration (brace Block Scope), allocated on Stack.

## The Stack

The Stack is part of the Program Virtual Memory. It is used to hold all necessary information about the active functions (at run-time).

➢      This includes its Local Variables (as well as parameters, return address, etc.)
➢      All Local Variables will take up space from the Stack.

Don't(s):
➢ Stack Memory serves program functionality (also known as "Call Stack").
➢ Superseding Stack Memory limitations can cause Stack Overflow:

```
void myFunction(){
    double dArr[100*100*100];
    /* possible stack overflow */
}
```

Auto storage duration (function Block Scope), allocated on Stack.

**Dynamic Allocation**

*Dynamic* Allocation is handled by the programmer (explicitly) at run-time.

➢ Programmer explicitly requests Allocation of a specific size memory from the system.

➢ System `return`s the starting address of the allocated memory chunk. This address can be used to access the allocated memory.

With *Dynamic* Allocation, the data structure/container sizes (e.g. array size) can adjust to the program needs at run-time.

➢ When memory is no longer required it should be explicitly Deallocated (free'd).

# Memory Allocation

## The Heap

The Heap is a special Virtual Memory part, "unused" by the Program functions and reserved for dynamically allocated objects / variables.

➢ All `new` Dynamically Allocated Variables will consume memory in the Heap.
➢ Future `new` allocations will fail if the memory becomes full.

Do(s):
➢ Significantly slower memory allocation than Stack.
➢ But Gbytes to work with – also called "Freestore".
➢ Use for "Big" data.
➢ Use for Dynamic data.

**The Heap**

The Heap is a special Virtual Memory part, "unused" by the Program functions and reserved for dynamically allocated objects / variables.

➢ All `new` Dynamically Allocated Variables will consume memory in the Heap.
➢ Future `new` allocations will fail if the memory becomes full.

Don't(s):
➢ Forget to check if the dynamic memory allocation request succeeded.
➢ Forget to explicitly free any memory explicitly allocated.
➢ Even with Gbytes available, an HD-resolution camera image capture while loop can clutter the computer memory in less than a minute, if the allocated memory for each image is not properly deallocated at each loop.
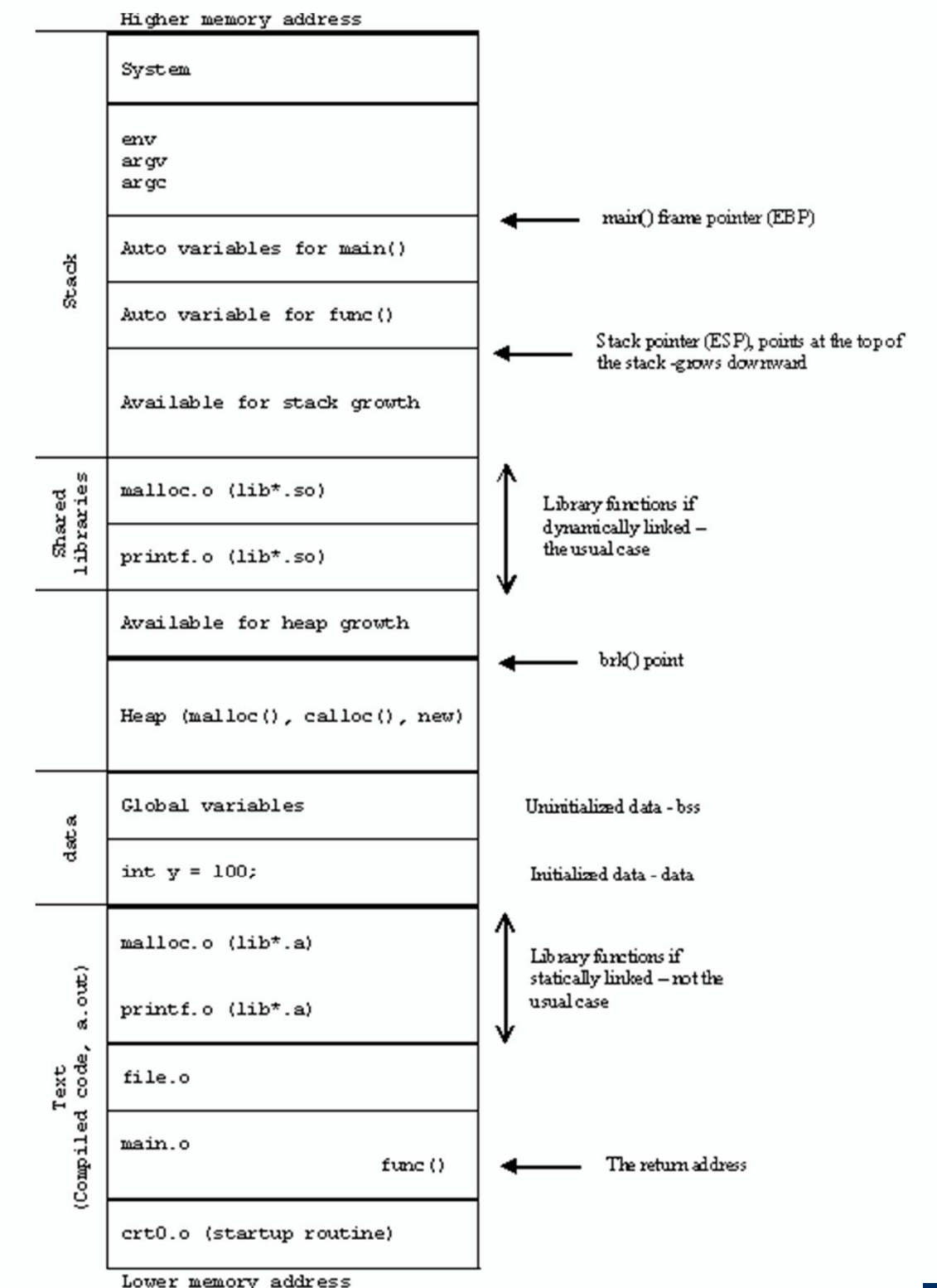
# Memory Allocation

## Program Memory

Overview of Program Memory Sections
(bottom-up from lower to higher Addresses)

➢ Compiled Text Section

➢ Data Section (BSS , Data)

➢ Dynamic Memory (**Heap**, Linked Libraries)

➢ **Stack** Memory

> Note:
> **NULL** is the `0` Address of the Virtual Memory.

# Dynamic Memory Allocation

**The Basics**

There is no named Object / Variable : All work is done on a Pointer-basis.

➢ Allocation reserves memory space.

➢ Address of reserved space is returned.

➢ Marked as "containing a specific data type" (**int**, **double**, **struct**, **class**, arrays, etc.)

Operator **new** dynamically Allocates memory space.
```
void * operator new (std::size_t count);
void * operator new [] (std::size_t count);
```

Operator **delete** can free-up this space (Deallocate memory) later on.
```
void operator delete (void * ptr);
void operator delete [] (void * ptr);
```

# Dynamic Memory Allocation

**The `new` (`[ ]`) *Expression***

Uses `operator new` (`[ ]`) to allocate memory space for the requested object / array type and size, and **return**s a Pointer-to (Address-of) the memory allocated.

➢ Pointer type as per requested type, marks what the memory contains.
➢ If sufficient memory is not available, the new operator returns `NULL` (let's say so for now…)
➢ The dynamically allocated object/array will persist through the program lifetime (memory will be reserved by it) until explicitly deallocated (i.e. by a `delete` Expression).

# Dynamic Memory Allocation

**The `new` (`[ ]`) *Expression***

Allocation of a single variable / object or an array of variables / objects.

Syntax:

`<type_id> * new <type_id_ctor>` (`[SIZE]` : **optional**)

Examples:

```
char * myChar_Pt = new char;
int * myIntArr_Pt = new int [20];
MyClass * myClass_Pt = new MyClass("mine",1,true);
MyClass * myClassArr_Pt = new MyClass [100];
```

> ➢ Simple-type variable.
> ➢ Simple-type variable array.
> ➢ Class-type instantiation in allocated memory.

Notes:

Before the assignment, the Pointer may or may not point to a "legitimate" memory.
After the assignment, the pointer points to a "legitimate" memory.

# Dynamic Memory Allocation

**The `new` (`[ ]`) *Expression***

Allocation of a single variable / object or an array of variables / objects.

Syntax:

`<type_id> * new <type_id_ctor>` (`[SIZE]` : **optional**)

Examples:

```
char * myChar_Pt = new char;
int * myIntArr_Pt = new int [20];
MyClass * myClass_Pt = new MyClass("mine",1,true);
MyClass * myClassArr_Pt = new MyClass [100];
```

➢ Simple-type variable.
➢ Simple-type variable array.
➢ Class-type instantiation in allocated memory.

Notes:

Before the assignment, the Pointer may or may not point to a "legitimate" memory.
After the assignment, the pointer points to a "legitimate" memory.

# Dynamic Memory Allocation

**The `new` (`[ ]`) *Expression***

Allocation of a single variable / object or an array of variables / objects.

Syntax:

`<type_id> * new <type_id_ctor>` (`[SIZE]` : **optional**)

Examples:

```
char * myChar_Pt = new char;
int * myIntArr_Pt = new int [20];
MyClass * myClass_Pt = new MyClass("mine",1,true);
MyClass * myClassArr_Pt = new MyClass [100];
```

➢ Simple-type variable.
➢ Simple-type variable array.
➢ Class-type instantiation in allocated memory.

Notes:

Before the assignment, the Pointer may or may not point to a "legitimate" memory.
After the assignment, the pointer points to a "legitimate" memory.

# Dynamic Memory Allocation

## The new ([ ]) Expression

```
1    //Program to demonstrate pointers and dynamic variables.
2    #include <iostream>
3    using std::cout;
4    using std::endl;


5    int main()
6    {
7        int *p1, *p2;

8        p1 = new int;
9        *p1 = 42;
10       p2 = p1;
11       cout << "*p1 == " << *p1 << endl;
12       cout << "*p2 == " << *p2 << endl;

13       *p2 = 53;
14       cout << "*p1 == " << *p1 << endl;
15       cout << "*p2 == " << *p2 << endl;
16       p1 = new int;
17       *p1 = 88;
18       cout << "*p1 == " << *p1 << endl;
19       cout << "*p2 == " << *p2 << endl;

20       return 0;
21   }
```
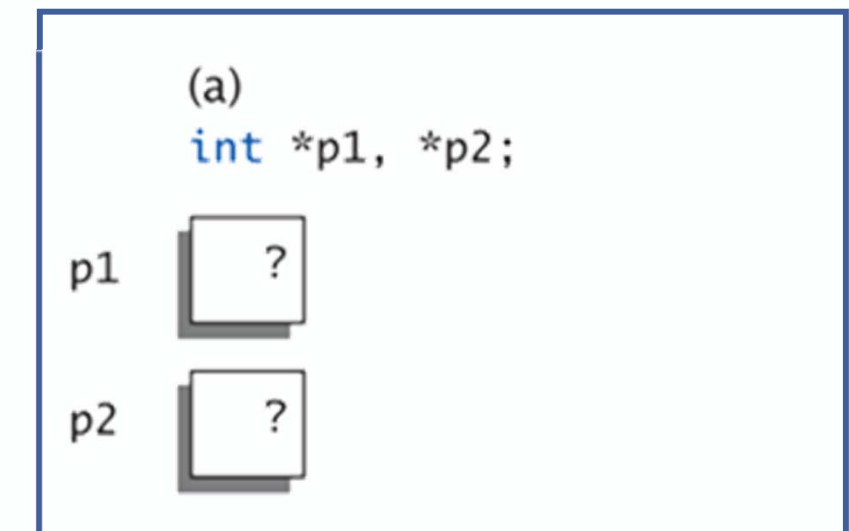
**SAMPLE DIALOGUE**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```

(a)
```
int *p1, *p2;
```

p1  [ ? ]

p2  [ ? ]

# Dynamic Memory Allocation

## The new ([ ]) Expression

```
1    //Program to demonstrate pointers and dynamic variables.
2    #include <iostream>
3    using std::cout;
4    using std::endl;


5    int main()
6    {
7        int *p1, *p2;

8        p1 = new int;
9        *p1 = 42;
10       p2 = p1;
11       cout << "*p1 == " << *p1 << endl;
12       cout << "*p2 == " << *p2 << endl;

13       *p2 = 53;
14       cout << "*p1 == " << *p1 << endl;
15       cout << "*p2 == " << *p2 << endl;
16       p1 = new int;
17       *p1 = 88;
18       cout << "*p1 == " << *p1 << endl;
19       cout << "*p2 == " << *p2 << endl;

20       return 0;
21   }
```
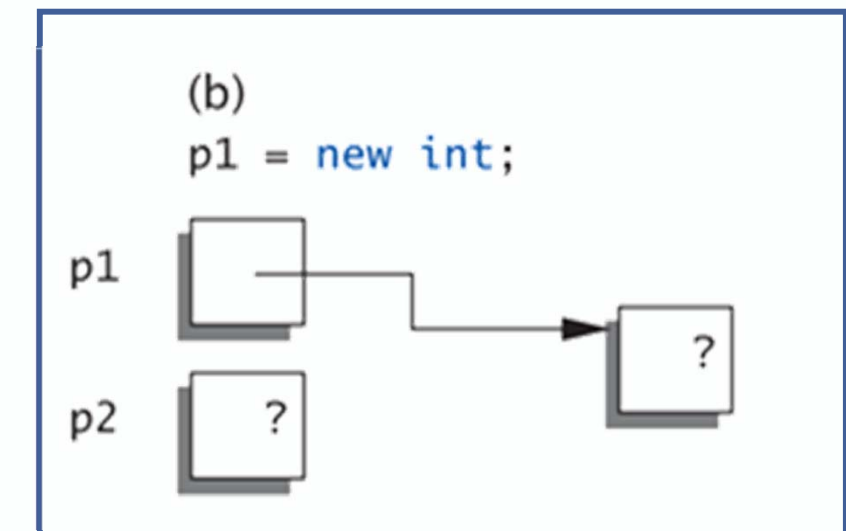
**SAMPLE DIALOGUE**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```



(b)
p1 = new int;

p1

p2     ?

?

# Dynamic Memory Allocation

## The new ([ ]) Expression

```
1    //Program to demonstrate pointers and dynamic variables.
2    #include <iostream>
3    using std::cout;
4    using std::endl;

5    int main()
6    {
7        int *p1, *p2;

8        p1 = new int;
9        *p1 = 42;
10       p2 = p1;
11       cout << "*p1 == " << *p1 << endl;
12       cout << "*p2 == " << *p2 << endl;

13       *p2 = 53;
14       cout << "*p1 == " << *p1 << endl;
15       cout << "*p2 == " << *p2 << endl;
16       p1 = new int;
17       *p1 = 88;
18       cout << "*p1 == " << *p1 << endl;
19       cout << "*p2 == " << *p2 << endl;

20       return 0;
21   }
```
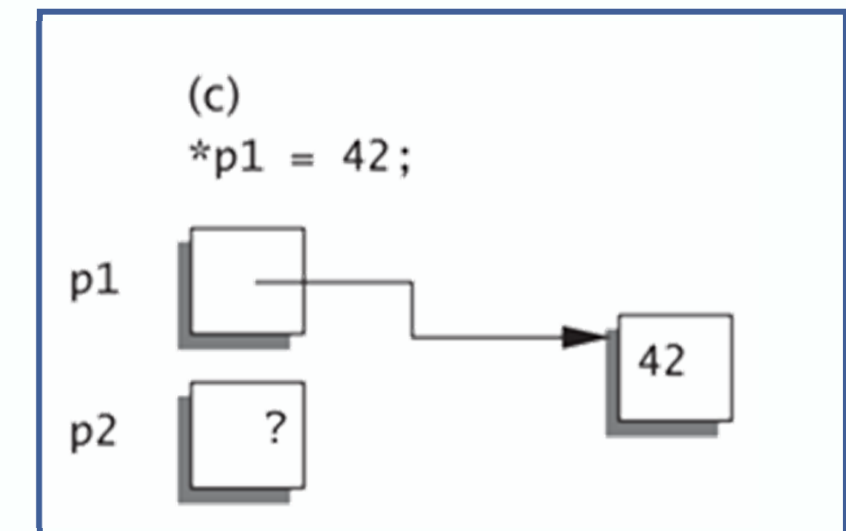
**SAMPLE DIALOGUE**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```

(c)
*p1 = 42;

p1

p2    ?

42

# Dynamic Memory Allocation

## The new ( [ ] ) Expression

```cpp
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;

5   int main()
6   {
7       int *p1, *p2;

8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;

13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
16      p1 = new int;
17      *p1 = 88;
18      cout << "*p1 == " << *p1 << endl;
19      cout << "*p2 == " << *p2 << endl;

20      return 0;
21  }
```
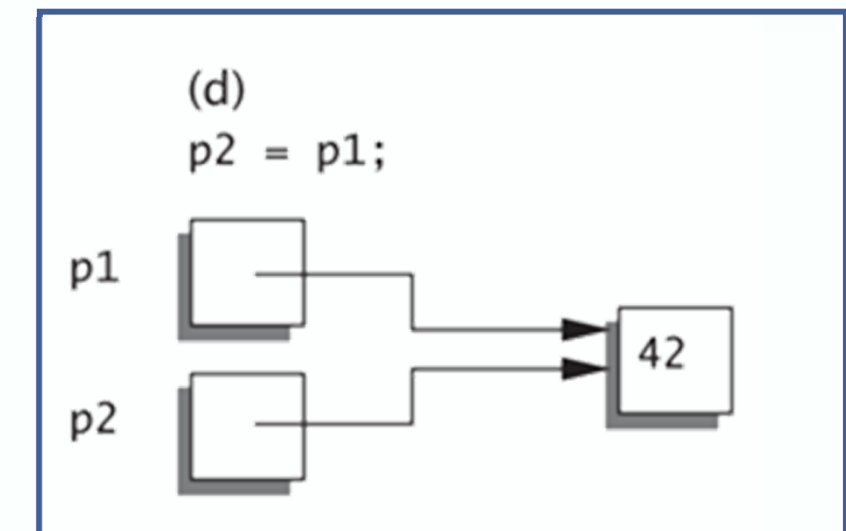
**SAMPLE DIALOGUE**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```

(d)
p2 = p1;

p1

p2

42

## The `new` (`[ ]`) *Expression*

```
1    //Program to demonstrate pointers and dynamic variables.
2    #include <iostream>
3    using std::cout;
4    using std::endl;


5    int main()
6    {
7        int *p1, *p2;

8        p1 = new int;
9        *p1 = 42;
10       p2 = p1;
11       cout << "*p1 == " << *p1 << endl;
12       cout << "*p2 == " << *p2 << endl;

13       *p2 = 53;
14       cout << "*p1 == " << *p1 << endl;
15       cout << "*p2 == " << *p2 << endl;
16       p1 = new int;
17       *p1 = 88;
18       cout << "*p1 == " << *p1 << endl;
19       cout << "*p2 == " << *p2 << endl;

20       return 0;
21   }
```
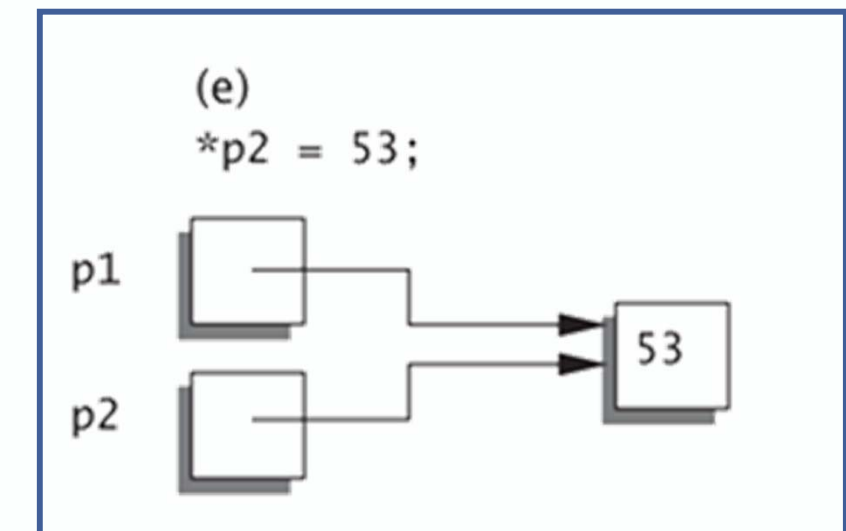
**SAMPLE DIALOGUE**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```



(e)
*p2 = 53;

p1

p2

53

# Dynamic Memory Allocation

## The new ([ ]) Expression

```cpp
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;

5   int main()
6   {
7       int *p1, *p2;

8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;

13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
16      p1 = new int;
17      *p1 = 88;
18      cout << "*p1 == " << *p1 << endl;
19      cout << "*p2 == " << *p2 << endl;

20      return 0;
21  }
```
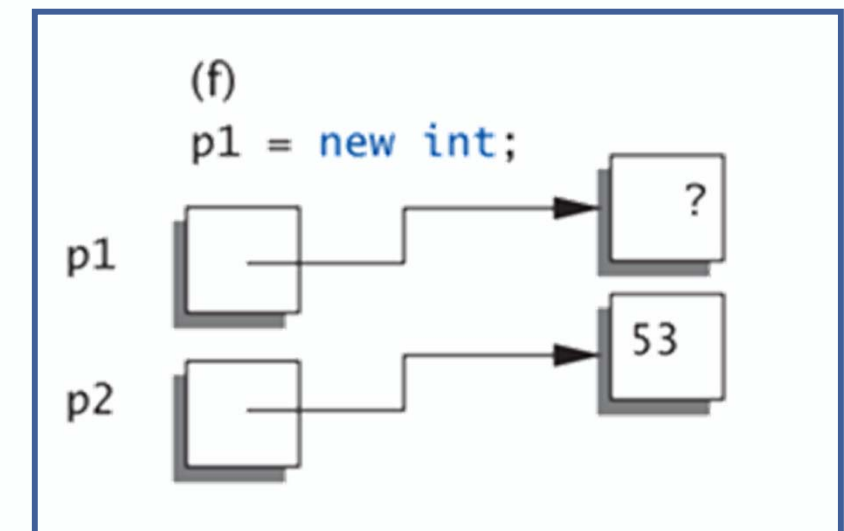
**SAMPLE DIALOGUE**

*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53



(f)
p1 = new int;

p1        ?

p2        53

# Dynamic Memory Allocation

## The new ([ ]) Expression

```cpp
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;

5   int main()
6   {
7       int *p1, *p2;

8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;

13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
16      p1 = new int;
17      *p1 = 88;
18      cout << "*p1 == " << *p1 << endl;
19      cout << "*p2 == " << *p2 << endl;

20      return 0;
21  }
```
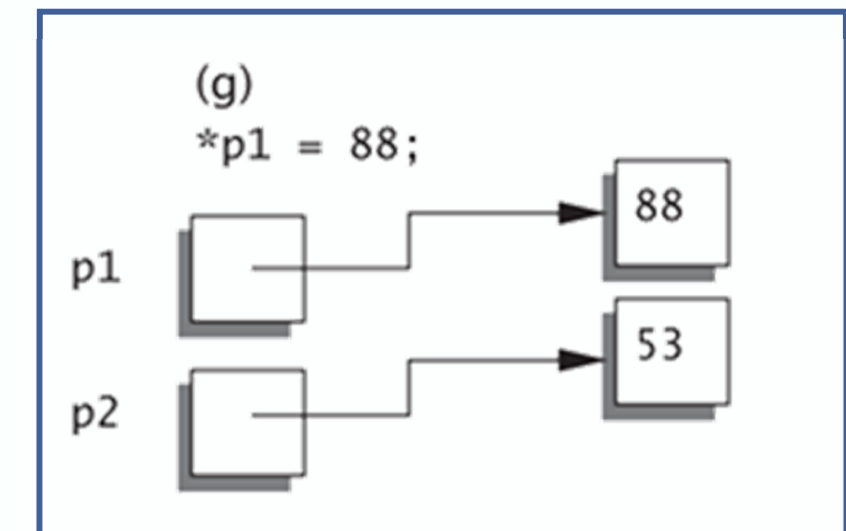
SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```



(g)
*p1 = 88;

p1

p2

88

53

**The `delete` (`[ ]`) *Expression***

Uses `operator delete` (`[ ]`) to Deallocate the object / array pointed-to by a pointer, which was the run-time result of a previous `new` Expression.

➢ Memory is free'd and returned to the Heap.
➢ Pointer is considered unassigned.
➢ If the value of the pointer is `NULL`, then `delete` has no effect (and it is safe to call).

## The `delete` (`[ ]`) *Expression*

Uses `operator delete` (`[ ]`) to deallocate the object / array pointed-to by a pointer, which was the run-time result of a previous `new` *Expression*.

➤ After `delete` is called on a memory region, it should no longer be accessed by the program.

Note: Otherwise, the result will be a Segmentation Fault.

➤ Convention is to set (/"mark") pointer to delete'd memory to `NULL`.

➤ Every `new` must have a corresponding `delete`.

Note: Otherwise, the program has memory leak.

➤ `new` and `delete` may not be in the same routine.

Note: But have to be properly sequenced during program execution.

# Dynamic Memory Allocation

## The `delete` (`[ ]`) *Expression*

Uses `operator delete` (`[ ]`) to deallocate the object / array pointed-to by a pointer, which was the run-time result of a previous `new` *Expression.*

➢ Called on a Pointer to dynamically allocated memory when it is no longer needed (only `new`'ed objects / variables can be `delete`'d).

```cpp
int globInt, globIntArr[5];
int main(){
    int locInt, locIntArr[5];
    int * int_Pt;
    int_Pt = &locInt;          delete intPt;
    int_Pt = &locIntArr;       delete [] intPt;
    int_Pt = &globInt;         delete intPt;
    int_Pt = &globIntArr;      delete [] intPt;
    return 0;
}
```

➢ Segmentation Fault
Trying to free non-dynamic (local variable, auto storage).

➢ Invalid Pointer Free
Memory address of global.

**The** `delete` **(** `[ ]` **)** *Expression*

Can delete a single object/variable or an array of objects/variables.

Syntax:

`delete` `<ptr_name>` **(** `[ ]` : optional**)**

Examples:

```
int * myInt_Pt = new int;
delete myInt_Pt;
char * myChar_Pt = new char [255];
delete [] myChar_Pt;

MyClass * myClass_Pt = new MyClass("mine", 1, true);
delete myClass_Pt;
MyClass * myClassArr_Pt = new MyClass [100];
delete [] myClassArr_Pt;
```

# Dynamic Memory Allocation

**The `delete` (`[ ]`) Expression**

Deleting an object/variable.

By-example:

```
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

| | Address | Value |
|---|---|---|
| **ptr** | FDE0 | |
| | FDE1 | |
| | FDE2 | |
| | FDE3 | |
| | 0EC4 | |
| | 0EC5 | |
| | 0EC6 | |
| | 0EC7 | |

# Dynamic Memory Allocation

**The delete ([ ]) Expression**

Deleting an object/variable.

By-example:

```cpp
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

| Address | Value |
|---------|-------|
| FDE0    | 0EC4  |
| FDE1    |       |
| FDE2    |       |
| FDE3    |       |
| 0EC4    |       |
| 0EC5    |       |
| 0EC6    |       |
| 0EC7    |       |

ptr

# Dynamic Memory Allocation

## The `delete` (`[ ]`) Expression

Deleting an object/variable.

By-example:

```cpp
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```
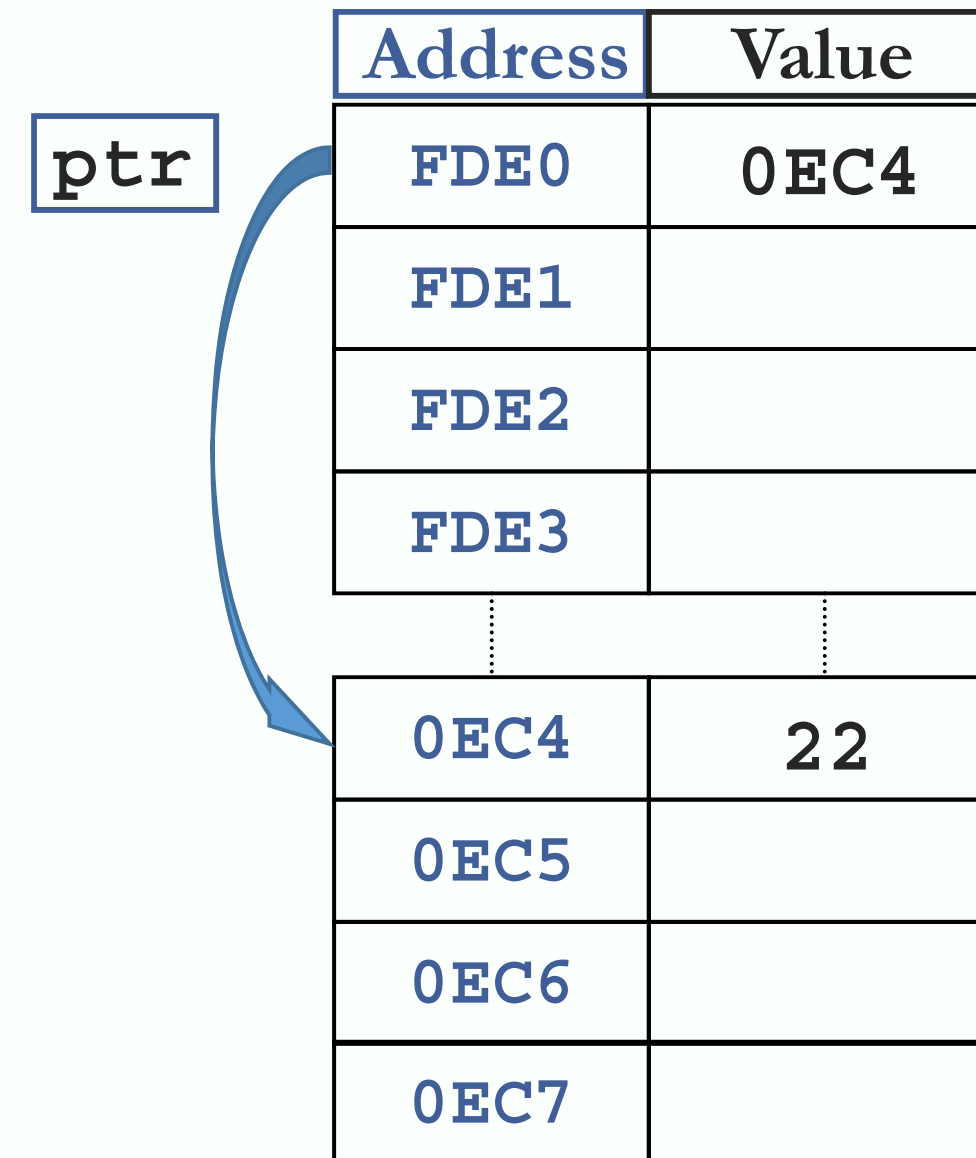
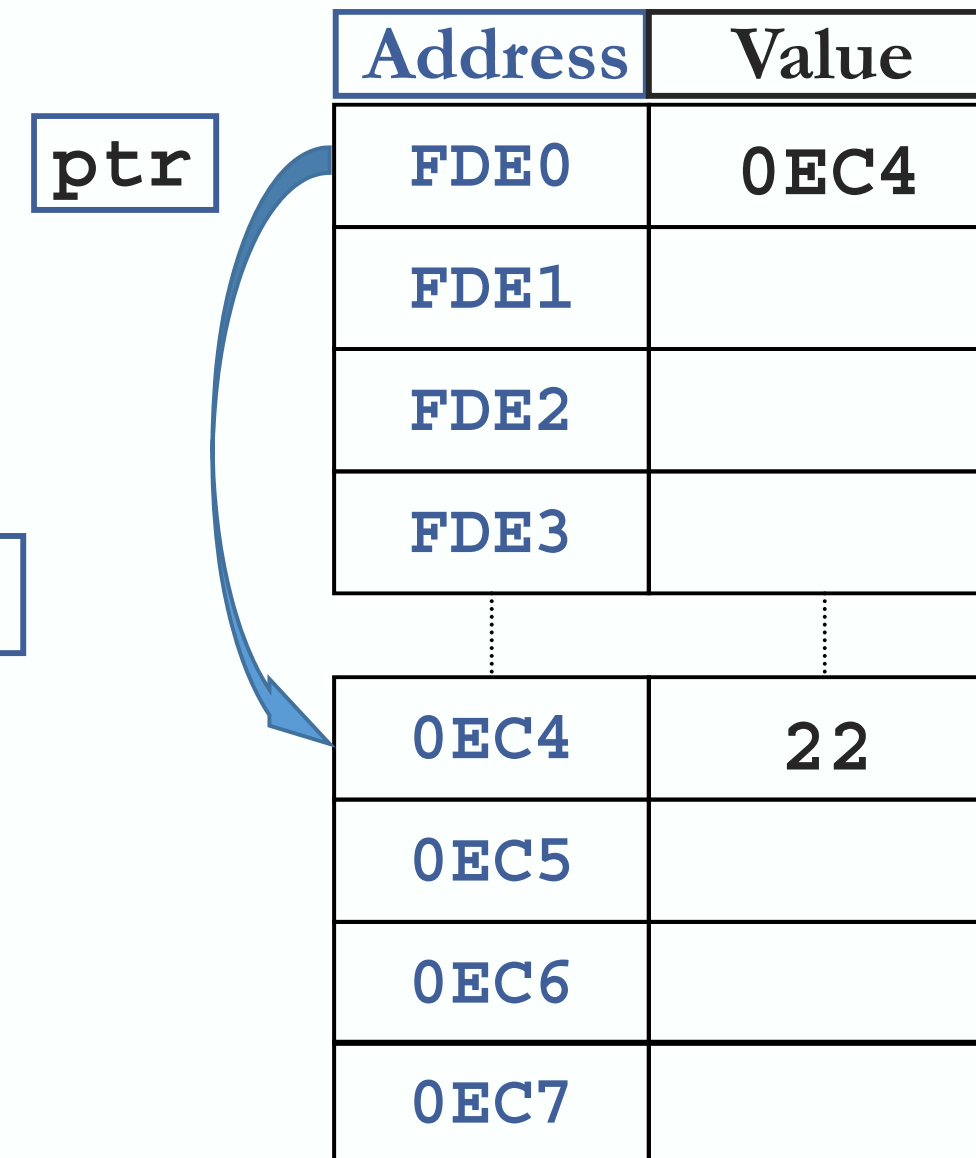| Address | Value |
|---------|-------|
| FDE0 | 0EC4 |
| FDE1 | |
| FDE2 | |
| FDE3 | |
| 0EC4 | 22 |
| 0EC5 | |
| 0EC6 | |
| 0EC7 | |

ptr
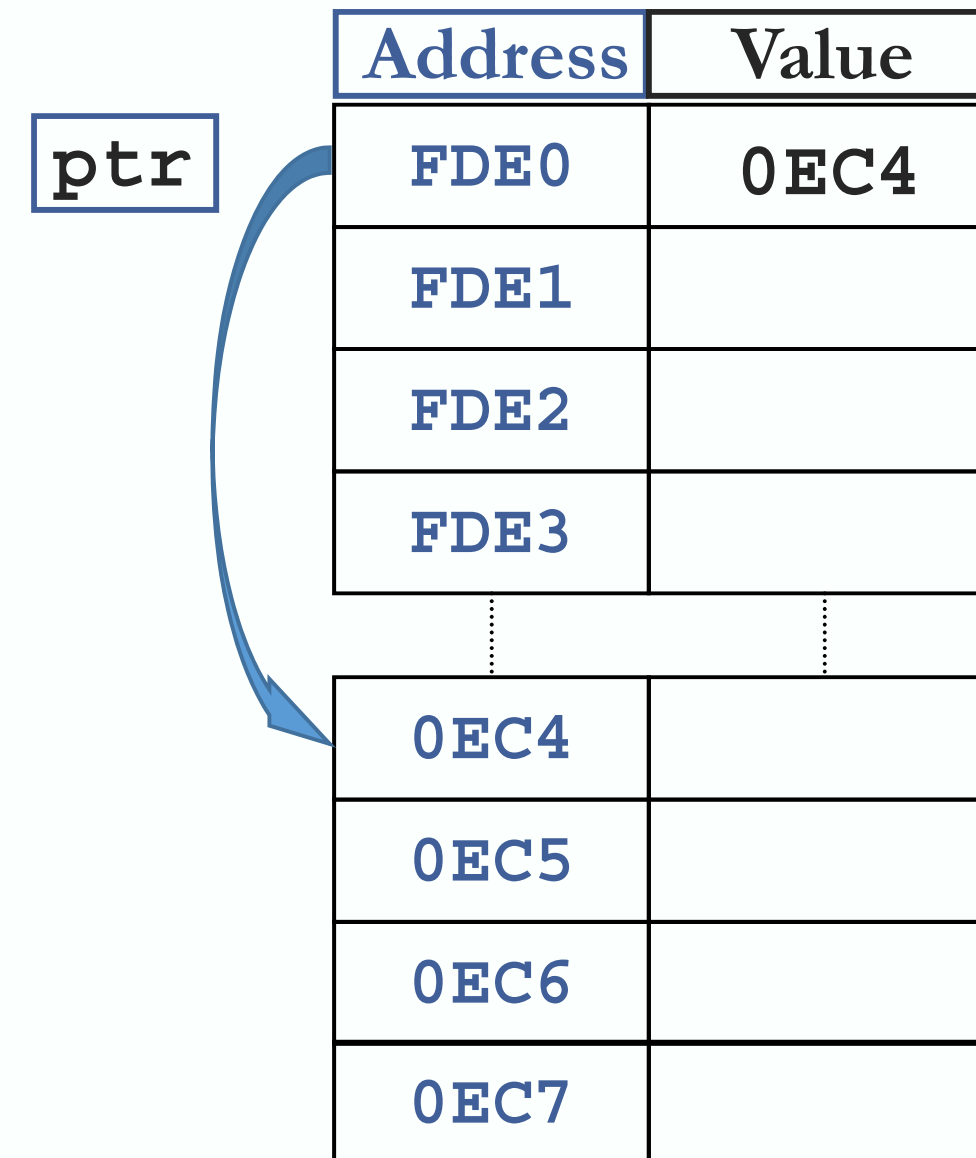
# Dynamic Memory Allocation

## The `delete` (`[ ]`) Expression

Deleting an object/variable.

By-example:

```cpp
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

Output: `22`

| Address | Value |
|---------|-------|
| FDE0 | 0EC4 |
| FDE1 | |
| FDE2 | |
| FDE3 | |
| 0EC4 | 22 |
| 0EC5 | |
| 0EC6 | |
| 0EC7 | |

ptr

# Dynamic Memory Allocation

## The delete ([ ]) Expression

Deleting an object/variable.

By-example:

```
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

| | Address | Value |
|---|---|---|
| ptr | FDE0 | ? |
| | FDE1 | |
| | FDE2 | |
| | FDE3 | |
| | 0EC4 | |
| | 0EC5 | |
| | 0EC6 | |
| | 0EC7 | |

# Dynamic Memory Allocation

## The delete ([ ]) Expression

Deleting an object/variable.

By-example:

```
int * ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

| Address | Value |
|---------|-------|
| FDE0    | 0     |
| FDE1    |       |
| FDE2    |       |
| FDE3    |       |
| 0EC4    |       |
| 0EC5    |       |
| 0EC6    |       |
| 0EC7    |       |

ptr

*Remember:* **Variable-Length Arrays (VLAs) are only an *Extension***

➢ And a non-Standard extension by GCC

```
const int start, end;

… // possible manipulation of start, end, etc.

double dNumbers[(start + end) / 2];
```

*Note:*
Non-constant expression used for size

By the GNU Compiler Collection – Online Docs
( http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html )

➢ Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C90 mode and in C++. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the block scope containing the declaration exits.

**Dynamically Allocated Array**

The `[IntExp]` Array-variant of the `new` Expression can be used to allocate arrays of objects/variables in Dynamic Memory.

```
char * myString = new char [255];
Car * myInventory = new Car [100];
```

Then `[IntExp]` Array-variant of the `delete` Expression can be used to indicate that an array of objects is to be Deallocated.

```
delete [] myString;
delete [] myInventory;
```

Note: Use Simple-variant or Array-variant properly (on an array).
      Otherwise the C++ Standard gives Undefined Behavior.

## Dynamically Allocated Array

By-Example:

```cpp
int * grades = NULL;
int numberOfGrades;

cout << "Enter the number of grades: ";
cin >> numberOfGrades;

grades = new int[ numberOfGrades ];

for (int i = 0; i < numberOfGrades; i++)
{   cin >> grades[i];   }

for (int j = 0; j < numberOfGrades; j++)
{   cout << grades[j] << " ";   }

delete [] grades;
grades = NULL;
```

**Dynamically Allocated Array**

By-Example:

```cpp
int * grades = NULL;
int numberOfGrades;

cout << "Enter the number of grades: ";
cin >> numberOfGrades;

grades = new int[ numberOfGrades ];

for (int i = 0; i < numberOfGrades; ++i)
{   cin >> grades[i];   }

for (int i = 0; i < numberOfGrades; ++i)
{   cout << grades[i] << " ";   }

delete [] grades;
grades = NULL;
```

Array size is determined during run-time!

**Dynamically Allocated 2D Array**

A two-dimensional array is an array of arrays (e.g. rows).

To dynamically allocate a 2D array, a double pointer is used.
➢ A pointer to a pointer.
```
<type_id> ** myMatrix;
```

Example: For a 2D integer array:
```
int ** intMatrix;
```

**Dynamically Allocated 2D Array**

Memory allocation the 2D array with **rows** rows and **cols** columns:

➢ Allocate of an array of pointers:

(these will be used to point to the sub-arrays – i.e. the rows)

```
int ** intMatrix = new int * [rows];
```

This creates space for rows number of Addresses (each element is an `int*`).

➢ Then allocate the space for the 1D arrays (i.e. the rows) themselves, each with a size of **cols**.

```
for (int i=0; i<rows; ++i)
    intMatrix[i] = new int [cols];
```

**Dynamically Allocated 2D Array**

The elements of the 2D array can still be accessed by the notation:

```
intMatrix[i][j];
```

Note:  The entire array is *NOT* (guaranteed to be) in contiguous space.
Unlike a statically allocated 2D array!

➢ Each row sub-array is contiguous in memory.
➢ But the sequence of rows is not.

`intMatrix[i][j+1]` is after `intMatrix[i][j]` in memory.

`intMatrix[i+1][0]` may be before or after `intMatrix[i][0]` in memory.

# Dynamic Arrays

## Dynamically Allocated 2D Array

By-Example:

```cpp
int rows, cols;
int ** intMatrix;

cin >> rows >> cols;
```

```cpp
a)  intMatrix = new int * [rows];
    for (int i=0; i<rows; ++i)
b)    intMatrix[i] = new int [cols];
```

Allocation:
a) Rows array of pointers first.
b) Each row sub-array then.

```cpp
c)  for (int i=0; i<rows; ++i)
      delete [] intMatrix[i];
d)  delete [] intMatrix;
```

Deallocation:
c) Each row sub-array first.
d) Rows array of pointers last.

**CS-202**

Time for Questions !