CS-202

Dynamic Memory (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 7th Project will be announced today Thursday 3/28.

- PASS Sessions will resume on Friday, get all the help you need!

Today's Topics

Memory Storage

- Automatic
- Static
- Dynamic

Program Memory

- Stack
- Heap

Program Memory Management

- Expression `new [ ]`
- Expression `delete [ ]`

Dynamic Memory Allocation

Remember: **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` (not quite anymore, but let's say so for right 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.

~~`MyClass * myClassArr_Pt = new MyClass [100]("mine",1,true);`~~ NO. Not allowed.

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 `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 to be considered *invalid*:
  (According to C++ Standard, 3.7.3.2/4 - the deallocation function will render invalid all pointers referring to all parts of deallocated storage)
- If the value of the pointer is `NULL`, then `delete` has no effect
  (and it is safe to call).

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*.

- After `delete` is called on a memory region, it should no longer be accessed by the program.
  Note: Otherwise, the result is Undefined Behavior (best hope is 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*

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*

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).

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

```
delete intPt;
delete [] intPt;
```

```
delete intPt;
delete [] intPt;
```

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

- Invalid Pointer Free
  Memory address of global.

Dynamic Arrays

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

- A C++ (non-Standard) extension by GCC

```
const int start, end;
... // possible manipulation of start, end, etc.
double dNumbers[(start + end) / 2];
```

Hint: Try compiling with **`-pedantic`**

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.

Dynamic Arrays

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:

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

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

grades = new int[numberOfGrades];

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

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

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

Dynamic Arrays

Dynamically Allocated Array

By-Example:

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

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

grades = new int[numberOfGrades];

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

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

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

Array size is determined during run-time!

See any problem here ?

Dynamic Arrays

Dynamically Allocated Array

By-Example:

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

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

grades = new int[numberOfGrades];

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

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

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

Array size is determined during run-time!

Have to check for **new** allocation success!

Dynamic Memory Allocation

The new ([ ]) Expression

Actually, **operator new** (**[ ]**) **throw**s !

Remember: Exceptions

If allocation fails, Expression **new** (**[ ]**) will throw a **std::bad_alloc** exception.

Proper syntax is:

```
try{
    char * myChar_Pt = new char [MAX_SIZE];
}
catch( const std::bad_alloc & ex ){
    /* handle exception ex by-const-reference … */
}
```

Note:
There is still however a non-throwing variant (has **noexcept** specification in C++11).

More about this later on in CS-202 …

(std::nothrow)

Dynamic Memory Allocation

The new ([ ]) Expression

Actually, **operator new** (**[ ]**) **throw**s !

Remember: Exceptions

If allocation fails, Expression **new** (**[ ]**) will throw a **std::bad_alloc** exception.

Proper syntax is:

```
try{
    char * myChar_Pt = new char [MAX_SIZE];
}
catch( const std::bad_alloc & ex ){
    /* handle exception ex by-const-reference … */
}
```

Note: Variant that is **noexcept** of **operator new** ([]) is invoked by the **new** *Expression*:

```
char * myChar_Pt = new (std::nothrow) char [MAX_SIZE];
```

And then you have to check whether the pointer is **NULL** …

Dynamically Allocated Array

By-Example (the proper way):

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

cin >> numberOfGrades;

try{
  grades = new int[numberOfGrades];
  for (size_t i = 0; i < numberOfGrades; ++i)
  {  cin >> grades[i];  }
  for (size_t i = 0; i < numberOfGrades; ++i)
  {  cout << grades[i] << " ";  }
  delete [] grades;
  grades = NULL;
}
catch(const std::bad_alloc & ex){
    /* handle exception ex by-const-reference … */
}
```

Exception handling.

Dynamic Arrays

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;
```

Dynamic Arrays

Dynamically Allocated 2D Array

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

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

Dynamic Arrays

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.

Dynamically Allocated 2D Array

By-Example:

```
a)  int rows, cols;
b)  int ** intMatrix;

    cin >> rows >> cols;

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

    for (size_t i=0; i<rows; ++i)
      delete [] intMatrix[i];
    delete [] intMatrix;
```

Required Variables:

a) Variables for number of rows, columns.

b) Dynamically allocated Array of pointers.

Dynamically Allocated 2D Array

By-Example:

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

cin >> rows >> cols;

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

for (size_t i=0; i<rows; ++i)
  delete [] intMatrix[i];
delete [] intMatrix;
```

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

Dynamically Allocated 2D Array

By-Example:

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

cin >> rows >> cols;

intMatrix = new int * [rows];

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

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

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

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
```

...

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
...
```

Deallocate rows array of pointers on exception (allocation failure).

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
…
```

Initialize: Set to **NULL**
(always *defensive* in the beginning - future exception handling *might* need so).

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
…
```

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
…
```

Deallocate all previously allocated row sub-arrays on exception (allocation failure for one)

Dynamically Allocated 2D Array

By-Example (the proper way):

```
try{
  intMatrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    intMatrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try
    { intMatrix[i] = new int [cols]; }
    catch(std::bad_alloc& ex){
      for (; i>=0; --i)
        delete [] intMatrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] intMatrix; }
...
```

Re-`throw` currently handled `std::bad_alloc` exception to continue handling of bad allocation.

Deallocate rows array of pointers on exception (allocation failure).

Memory Handling

Memory Leaks

When creating objects with Dynamic Memory allocation, access is provided through the **`prvalue`** pointer of the Expression **`new`** (**`[ ]`**).

- I.e. the pointer (of requested type) to the newly allocated memory.
- To keep track and access in the future, this is stored to a pointer variable.

Reassigning that pointer, letting it go out of scope without maintaining its value, etc. without first **`delete`**ing the memory it used to pointed to, is called a Memory Leak.

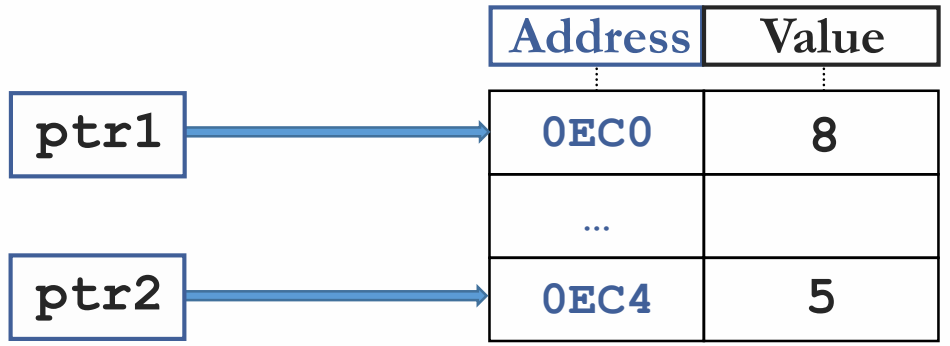
- Unless explicitly instructed to be Deallocated (by a **`delete`** (**`[ ]`**) Expression), that memory part will remain reserved.
- Memory leaks result in loss of available memory space.

Memory Handling

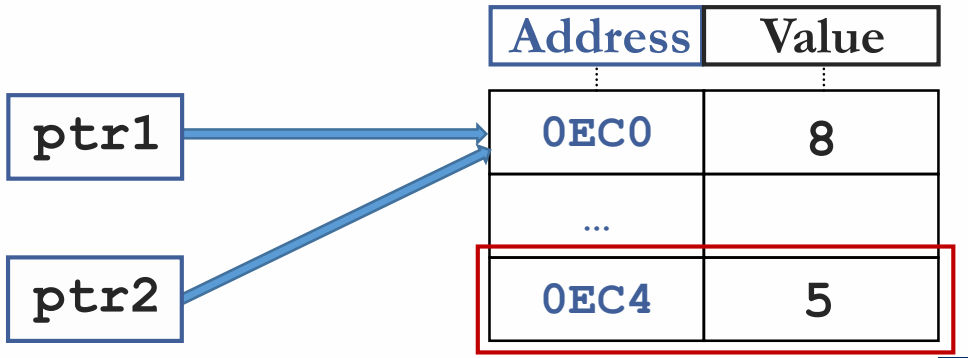
Memory Leak

A pointer that points to Dynamic Memory that has previously been Deallocated.

```
int * ptr1 = new int;
int * ptr2 = new int;
*ptr1 = 8;
*ptr2 = 5;
```




```
ptr2 = ptr1;
```



Memory Handling

Inaccessible Object

An *Unnamed* Object that was created by Expression `new ( [ ] )` and which has been left with no pointer to it by the programmer.

- A logical error.
- A common cause of Memory Leaks.

Memory Handling

Dangling Pointer

A pointer that points to Dynamic Memory that has previously been Deallocated.

- Allocation and Deallocation properly implemented, but pointer never set to `NULL` to satisfy convention.
- Could also happen with uninitialized pointer.

Note: Dereferencing a dangling pointer is undefined behavior per the C++ standard:

An **`lvalue`** of a non-function, non-array type T can be converted to an **`rvalue`** … If the object to which the **`lvalue`** refers is not an object of type T and is not an object of a type derived from T, or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.

Memory Handling

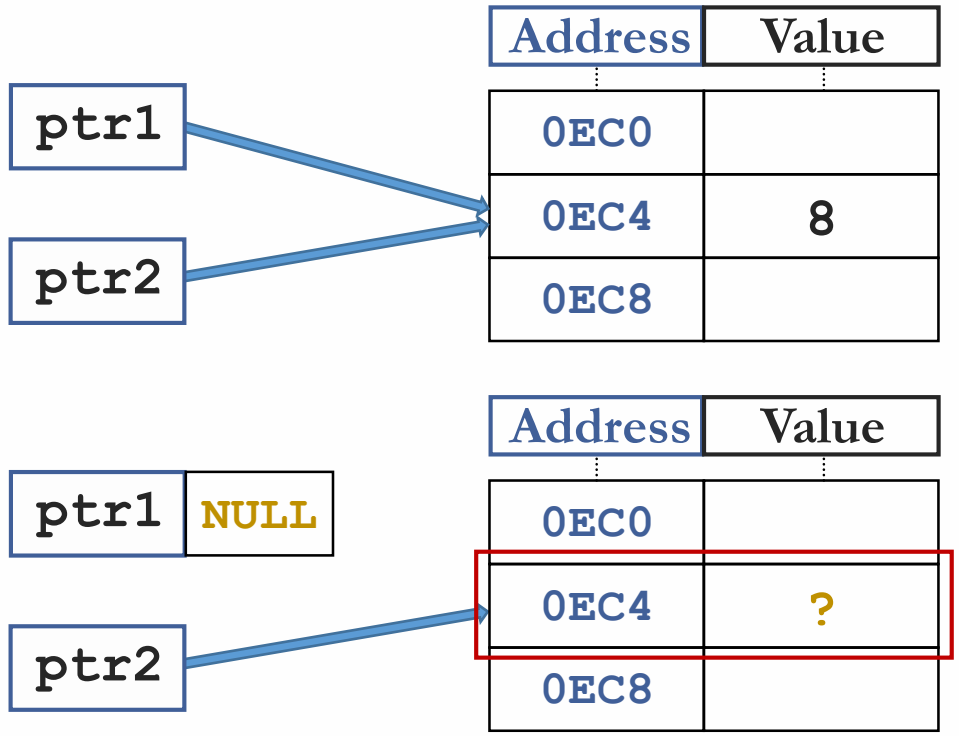
Dangling Pointer

A pointer that points to Dynamic Memory that has previously been deallocated.

```
int * ptr1 = new int;
int * ptr2;

*ptr1 = 8;
ptr2 = ptr1;
```

```
delete ptr1;
ptr1 = NULL;
```



Memory Handling

Dynamically Allocated 2D Array (By-Demonstration)

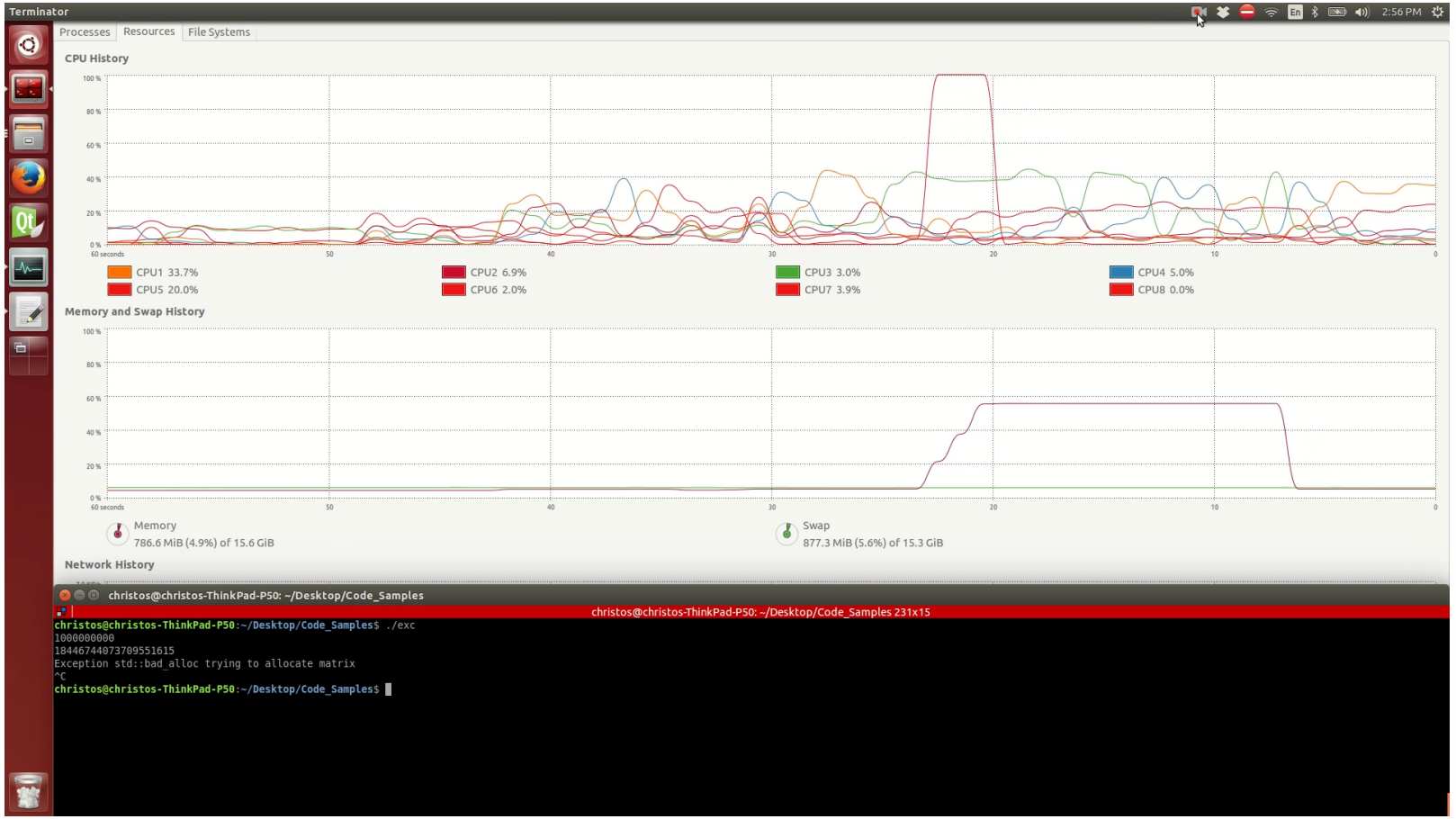
Type:
`int`

rows:

1,000,000,000

cols:

18,446,744,073,709,551,615



Memory Handling

Dynamically Allocated 2D Array (By-Demonstration)

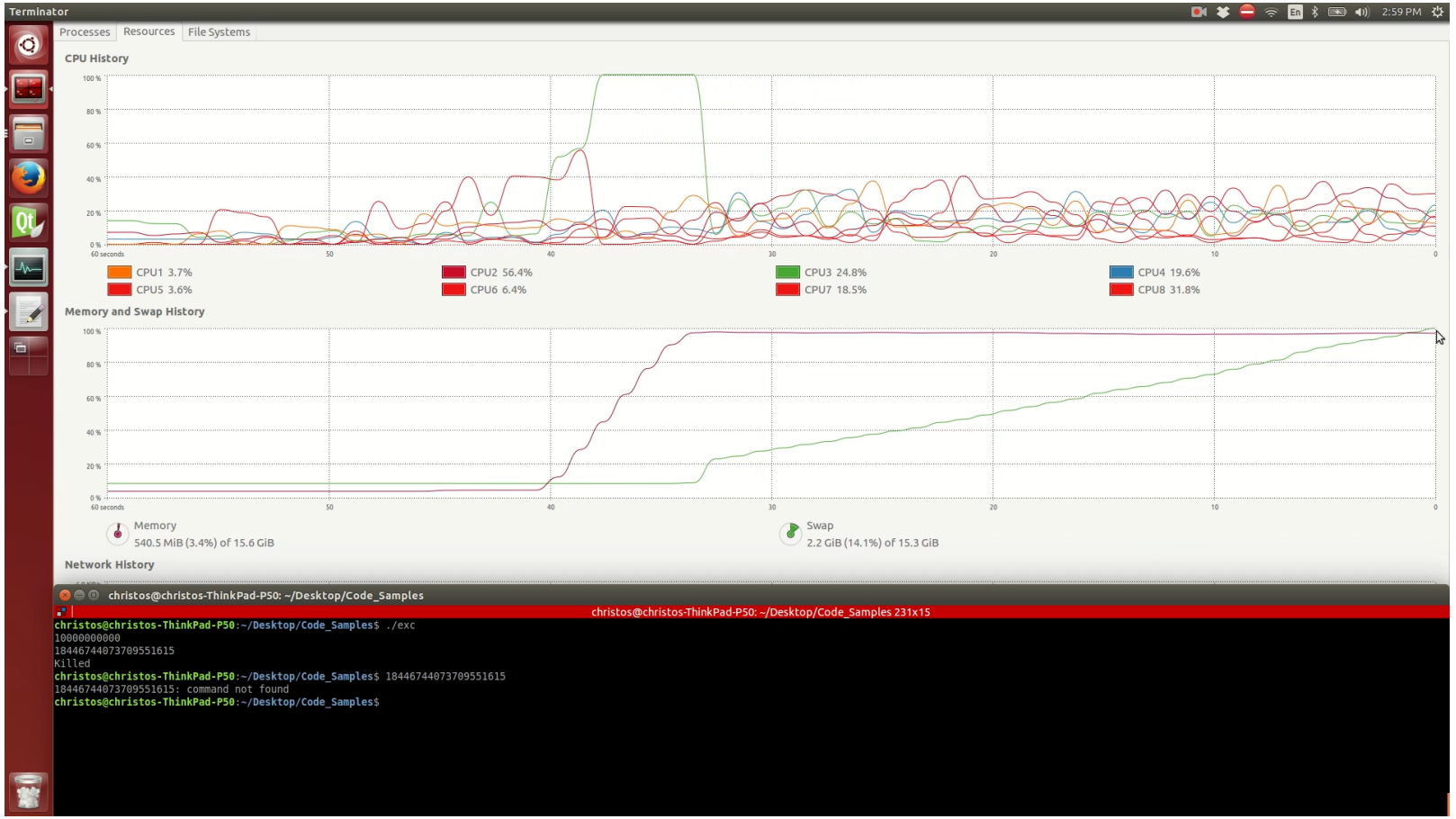
Type:
`int`

rows:

10 * 1,000,000,000

cols:

18,446,744,073,709,551,615



CS-202

Time for Questions !