# CSE 165/ENGR 140 Intro to Object Orient Program

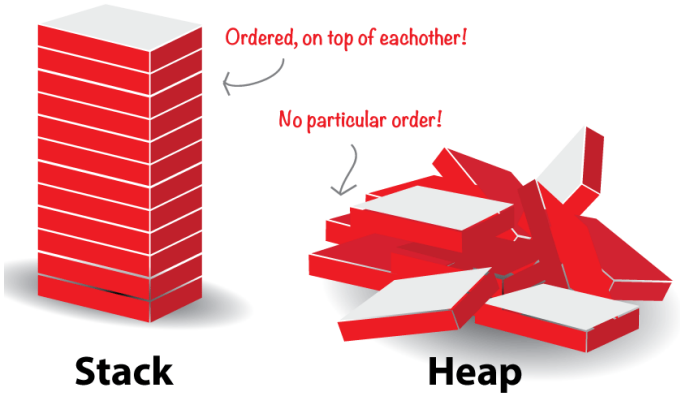
## Lecture 15 – Dynamic Object Creation

# Announcement

- Reading assignment
  - Ch. 10

# Object creation

- Object creation has two main parts:
  - allocates needed memory to store object(s)
    - memory can be in static (global) area
    - memory can be allocated on the stack (temporary variables)
    - memory can be allocated dynamically from the “heap”
  - calls constructor(s) to initialize the storage



# *new* and *delete*

- Low-level memory allocation
  - malloc() : allocates a given number of bytes
  - free() : releases allocated memory

- Object allocation in C++
  - new
    - allocates needed memory to hold object(s)
    - automatically calls constructors
  - delete
    - releases memory used for the object
    - automatically calls destructors
  - Important for proper object initialization!

# Examples

```
// standard way of using malloc/free:
int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj)); // no obj constructors are called here
  if ( !obj ) error("No Memory!"); // check if call was successful
  obj->initialize(); // requires explicit obj initialization
  ...
  obj->destroy(); // prepare obj for destruction
  free(obj); // free used memory
}
```

```
// using new/delete:
int main() {
  Obj* obj1 = new Obj; // default constructor is called!
  Obj* obj2 = new Obj(2); // integer constructor called
  // we will see how to check success of allocation later
  ...
  delete obj1; // calls destructor and frees used memory
  delete obj2; // calls destructor and frees used memory
}
```

# Delete with inheritance

```
class Shape {
  public:
    virtual ~Shape() {cout<<"~Shape"<<endl;};
};
class Rectangle : public Shape {
  public:
    virtual ~Rectangle() {cout<<"~Rectangle"<<endl;};
};
class Square : public Rectangle {
  public:
    virtual ~Square() {cout<<"~Square"<<endl;};
};

int main() {
  Square* s = new Square();
  delete s;
}
```

# *new* and *delete* with arrays

```
class A {
 public:
  int i;
  A ( int k ) { i=k; cout<<"A"<<i<<" Integer Constructor\n"; }
  A () { i=0; cout<<"A"<<i<<" Default Constructor\n"; }
 ~A () { cout<<"A"<<i<<" Destructor\n"; }
};

int main() {

A* a1 = new A[5]; // creates 5 objects initialized with default constructors
delete[] a1;      // 5 destructors will be called (note that you have to use
                  // "delete[]")

A* a2 = new A(5); // creates 1 object initialized with integer constructor
delete a2;        // 1 destructor is called

return 1;
}
```

# Checking successful allocation

- Running out of memory
  - When there is no more memory available, your program will “throw an exception”.
  - You can customize this behavior by defining your own “handler”.

# Examples

```
// The following is WRONG:
p = new X;
if (!p) //not to be used under ISO compliant compilers
{
 cout<<"allocation failure!"
 exit(1);
}


// A standard compliant version of this code should look like this:
try
{
 p = new DerivedWind;
}
catch (std::bad_alloc &ba)
{
 cout<<"allocation failure!"
 //...additional cleanup
}
```

# Examples

```
//: C13:NewHandler.cpp
// Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count << " allocations!" << endl;
  // For example here you could free some "reserve" memory
  // previously allocated for emergencies etc
}

int main() {
  set_new_handler ( out_of_memory ); //handler function when new fails
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
}
```

# Wake up!

- https://youtu.be/2RnW6Oxo45M

# Overloading

- ***new*** and ***delete*** can be overloaded
  - globally
  - per class
    - special versions exist for single object
    - for creating an array of objects

- constructors and destructors are always called
  - your overloaded method/function just decides how to allocate/free memory.

# Overloading global *new* & *delete*

```
//: C13:GlobalOperatorNew.cpp
#include <cstdio>    //Do not use iostream because cout calls new to
#include <cstdlib>     // allocate memory.
using namespace std;
void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}
void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class S {
  int i[100];
public:
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};
```

```
int main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  S* s = new S;
  delete s;
  puts("creating & destroying S[3]");
  S* sa = new S[3];
  delete []sa;
}
```

# Overloading global *new* & *delete*

***Output:***

```
creating & destroying an int
operator new: 4 Bytes
operator delete
creating & destroying an s
operator new: 400 Bytes
S::S()
S::~S()
operator delete
creating & destroying S[3]
operator new: 1204 Bytes
S::S()
S::S()
S::S()
S::~S()
S::~S()
S::~S()
operator delete
```

# Overloading *new* & *delete* for a class

```
class A {
 public:
  int i;
  A ( int k ) { i=k; cout<<"A"<<i<<" Integer Constructor\n"; }
  A () { i=0; cout<<"A"<<i<<" Default Constructor\n"; }
 ~A () { cout<<"A"<<i<<" Destructor\n"; }

  void* operator new (size_t sz) {
    cout << "==>new: " << sz << " bytes" << endl;
    return ::new char[sz]; // call global new
  }
  void operator delete(void* p) {
    cout << "==>delete" << endl;
    ::delete []p; // call global delete
  }
  void* operator new[](size_t sz) {
    cout << "==>new[]: " << sz << " bytes" << endl;
    return ::new char[sz]; // call global new
  }
  void operator delete[](void* p) {
    cout << "==>delete[]" << endl; // call global delete
    ::delete []p;
  }
};
```

# Overloading *new* & *delete* for a class

```
// overloaded new/delete can be tested with:

int main() {

  A* a1 = new A[5]; // creates 5 objects initialized with default
          // constructors
  cout<<a1->i<<endl;
  delete[] a1;      // 5 destructors will be called (note that you have to
                    // use "delete []")

  A* a2 = new A(5); // creates 1 object initialized with integer constructor
  cout<<a2->i<<endl;
  delete a2;        // 1 destructor called

  return 0;
}
```

# Overloading *new* & *delete* for a class

***Output:***

```
==>new[]: 24 bytes
A0 Default Constructor
A0 Default Constructor
A0 Default Constructor
A0 Default Constructor
A0 Default Constructor
0
A0 Destructor
A0 Destructor
A0 Destructor
A0 Destructor
A0 Destructor
==>delete[]
==>new: 4 bytes
A5 Integer Constructor
5
A5 Destructor
==>delete
```

# Explicit constructor/destructor calls

- explicit calls to constructors and destructors are possible, ex:

```
x.X::X();   or   xpt->X::X();
x.X::~X();   or   xpt->X::~X();
```