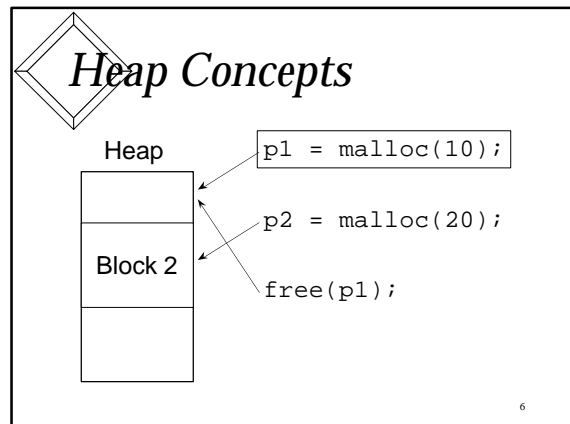
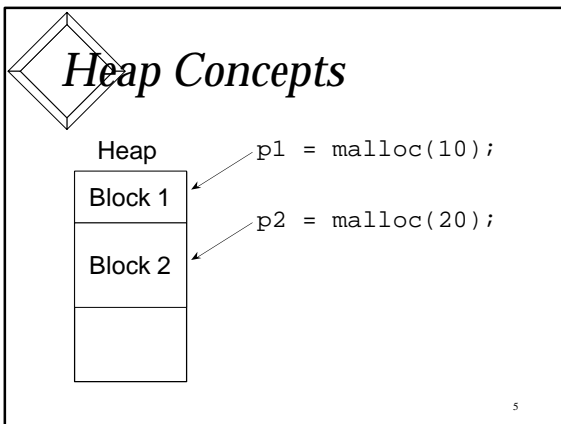
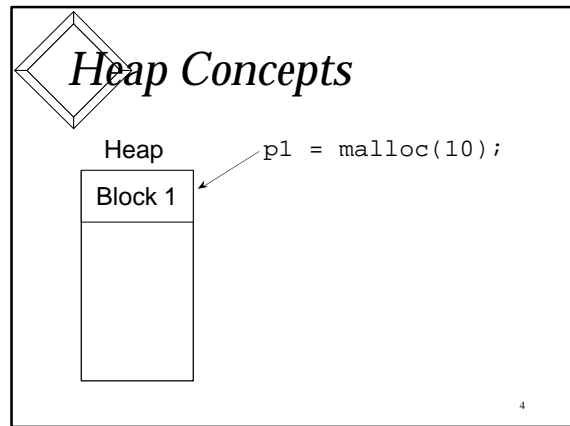
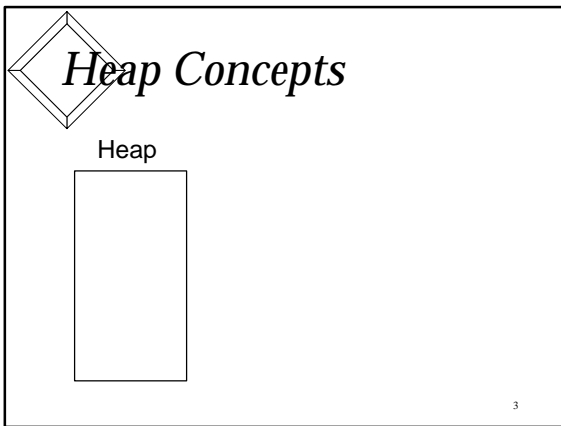
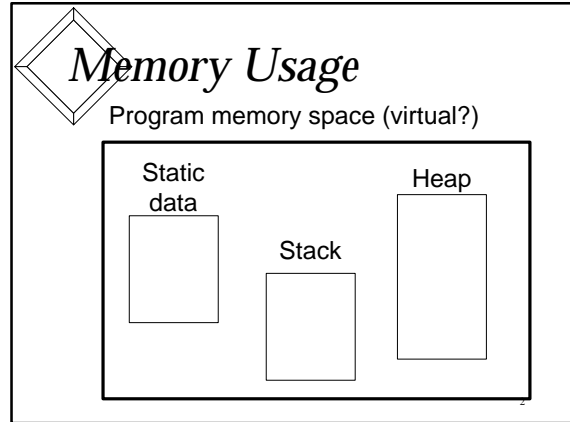


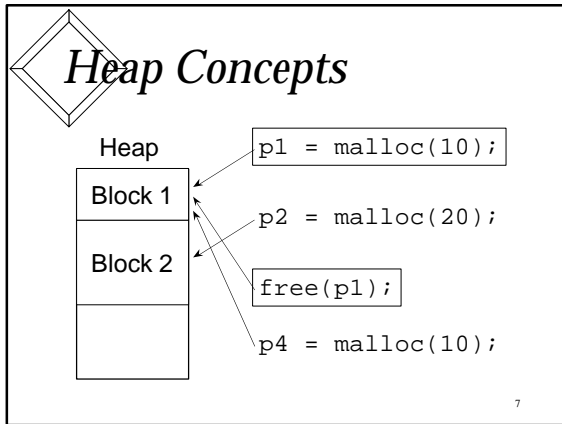
CS481 - Object-Oriented Programming

Types of Memory

- ❖ **Static**
 - Allocated for entire program run time
- ❖ **Automatic (stack)**
 - Allocated for life of program block
- ❖ **Dynamic (heap)**
 - Allocated under program control



CS481 - Object-Oriented Programming



Memory Leak Reminder

- ❖ Dynamic memory allocated
 - From heap: malloc, calloc, realloc
- ❖ Must be deallocated
 - To heap: free
- ❖ If no deallocation, memory lost
 - Heap grows; consumes address space

Memory Usage

- ❖ Static (one copy; known at compile)
- ❖ Stack (associated with block)
 - Possibly multiple copies (recursion)
- ❖ Heap (unspecified number of copies)
 - Flexible allocation
 - Usage depends on program flow

Dynamic Objects in C++

- ❖ Created on the heap
- ❖ Must allocate memory
 - Could use malloc(...)?
- ❖ But also need constructor
 - Can't call constructor directly
 - ◆ Same problem for destructor

C++ Heap Operators

- ❖ Must be part of language
 - So compiler can invoke constructor and destructor
 - Can't be just library routines
- ❖ Operator new - allocate/construct
- ❖ Operator delete - destroy/free

new & delete

```
int main()
{
    Complex* pc1; // pointers!!
    Complex* pc2;
    pc1 = new Complex(1.0, 2.0);
    pc2 = new Complex;
    ...
    delete pc1;
    delete pc2;
}
```

CS481 - Object-Oriented Programming

new & delete

Note: no type cast necessary

```
int main()
{
    Complex* pc1; // pointers!!
    Complex* pc2;
    pc1 = new Complex(1.0, 2.0);
    pc2 = new Complex;
    ...
    delete pc1;
    delete pc2;
    char* cp = (char*) malloc(10);
}
```

13

new & delete

Automatic sizing

```
int main()
{
    Complex* pc1; // pointers!!
    Complex* pc2;
    pc1 = new Complex(1.0, 2.0);
    pc2 = new Complex;
    ...
    delete pc1;
    delete pc2;
    char* cp = (char*) malloc(10);
}
```

14

new & delete

Use any constructor (even default)

```
int main()
{
    Complex* pc1; // pointers!!
    Complex* pc2;
    pc1 = new Complex(1.0, 2.0);
    pc2 = new Complex;
    ...
    delete pc1;
    delete pc2;
}
```

15

What About ... ?

```
void f()
{
    Complex* pc1;
    pc1 = new Complex;

    // forgot to delete it!

    // delete pc1;
}
```

16

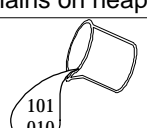
What About ... ?

```
void f()
{
    Complex* pc1;
    pc1 = new Complex;

    // forgot to delete it!

    // delete pc1;
}
```

Memory leak; pc1 goes out of scope and disappears, but dynamic object remains on heap.



17

And ... ?

```
void f()
{
    Complex c1(1.0, 2.0);
    Complex* pc1 = &c1;

    delete pc1;
}
```


18

CS481 - Object-Oriented Programming

And ... ?

```
void f()
{
  Complex c1(1.0, 2.0);
  Complex* pc1 = &c1;


  delete pc1;
}
```



Not good! This attempts to free a local (automatic) variable that is on the stack, not the heap.

19

Dynamic Arrays



- ❖ In C, use malloc/calloc
 - array = (int*) calloc(n, sizeof(int));
- ❖ In C++, variant of new
 - Array = new Complex[n];
 - ◆ Note: default constructor used

20

Array Deletion

```
int main()
{
  Complex* pc;
  Complex* ac;
  pc = new Complex;
  ac = new Complex[10];
  ...
  delete pc;
  delete ac;
}
```

Each delete calls destructor for one object. What about other nine array elements?

21

Array Deletion

```
int main()
{
  Complex* pc;
  Complex* ac;
  pc = new Complex;
  ac = new Complex[10];
  ...
  delete pc;
  delete[] ac;
}
```

Solution: special form of delete operator for arrays; calls destructor for each object.

22

Why delete[]?

- ❖ How many destructors?
 - Memory block size is known
 - But # of objects is not (why not?)
- ❖ Storage efficiency
 - Don't want to require "count" value to be stored for single objects

23

Why delete[] a; ?

- ❖ Consider the following:
 - a
 - &a
 - &a[0]
 - &a[]
 - a[]

24

CS481 - Object-Oriented Programming

Why delete [] a;

- ❖ Consider the following:
 - a
 - &a
 - &a[0]
 - &a[]
 - a[]


all produce an address to the start of the array

25

Dynamic Arrays

- ❖ Match new[] with delete[]
- ❖ Calls all destructors
- ❖ What about built-in types?
 - s = new char[strlen(S)+1];
 - delete s;
 - OK(?), but better to be consistent! - []

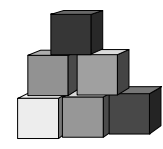
Allocation Failure



- ❖ What if new fails?
 - Not enough dynamic memory
- ❖ Calls “new handler” (MS Win?)
 - By default, throws exception
 - User-defined: set_new_handler()
 - ◆ Recover? Return NULL?

27

Custom Allocation



- ❖ Standard allocator
 - Like malloc
 - Manages common heap (varying sizes)
- ❖ Special allocator (& deallocator?)
 - More efficient (e.g., pre-allocations)
 - Overload new (global or per-class)


28

Re-new??

- ❖ What about realloc?
 - Change size of allocated block
 - Perhaps without moving it
- ❖ No equivalent in C++
 - Can do new[], copy, delete[]
 - ◆ Not as efficient?

29

Peaceful Coexistence



- ❖ C heap routines
 - malloc, calloc, realloc, free
- ❖ C++ heap operators
 - new, new[], delete, delete[]
- ❖ Don't mix the methods
 - On one memory block

30

CS481 - Object-Oriented Programming

