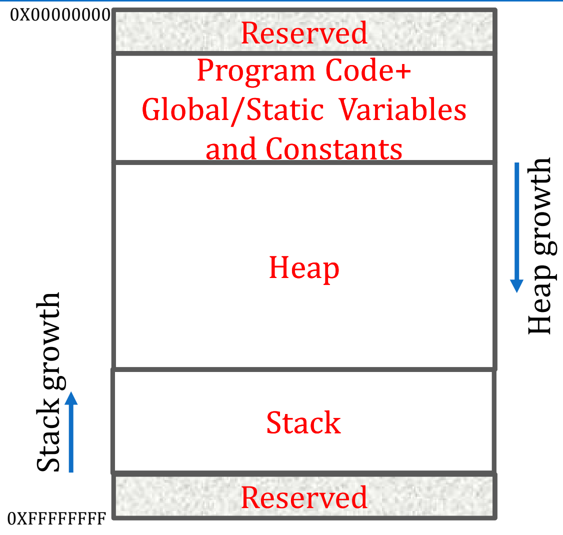
Data Structures
Computers 303B
Dynamic Allocation and Pointers

Prof. Dr. Khaled Fouad Elsayed

Memory Map of a Program

2

- Reserved Sections: used by operating system ➔ Fixed
- Code section:
  - program code (text): read-only
  - global/static (data and bss): read/write
  - Fixed size
  - Can be examined with the programs: size and objdump (available with gcc binutils)
- Stack:
  - grows from high to low addresses.
  - Stores local variables, function parameters, and call activation records for function calls
  - Dynamic read/write

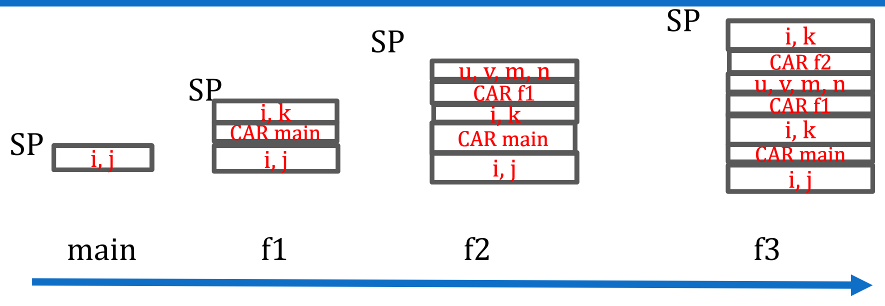


Heap

- The (huge) memory between code section and stack.
- Variable Size (as Stack is variable)
- Used for Dynamic Allocation during run-time.

Stack Growth

```
main() {
     int i, j;
     f1();
     .... Rest of program
}

f1(){
    int i, k;
    f2(I,j);
}

f2(int u, v){
    int m, n;
    n = f3(m);
}

int f3(int i)
{
    int k = i*i;
    return (k);
}
```



When f3 starts to return to main reverse action on stack occurs

Heap Usage

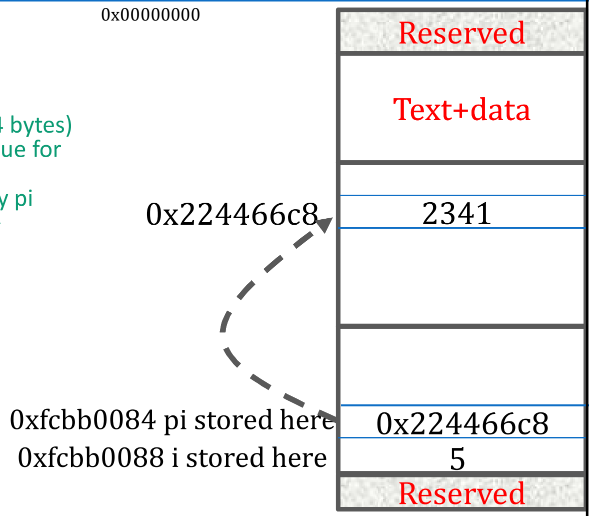
5

- Done via two operators only:
  - new
  - delete
  - Same for all data types

Example

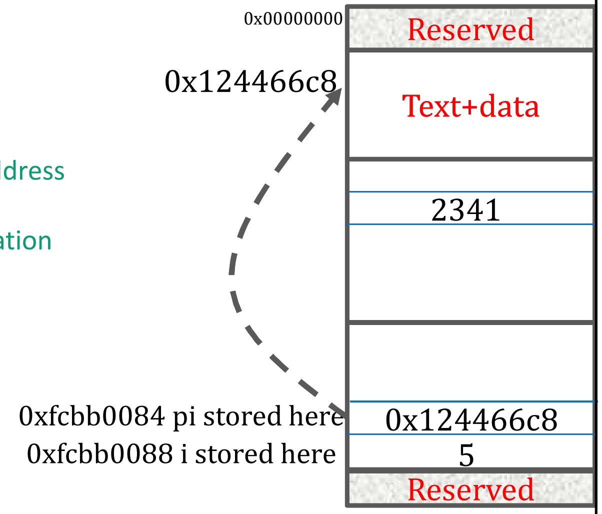
6

```
main() {
    int i = 5;
    int *pi; //defines pointer to int

    pi = new int; //Ask the heap library to reserve space for an int (4 bytes)
     assert(pi != NULL); //don't assume it will work. Null is illegal value for
pointer as it is in reserved section
     *pi = 2341; //assign the value 2345 to the location pointer to by pi
  // Alternatively if (pi != NULL) *pi = 2341; else { handle the error}

   for (*pi=0; *pi < 10; (*pi)++)
        cout << *pi;

   delete pi; //Return location allocated to pi back to heap

   * pi = 8;   //DISASTER: Don't ever use after de-allocation.
     // Will still compile

    // Only delete when no longer needed
}
```



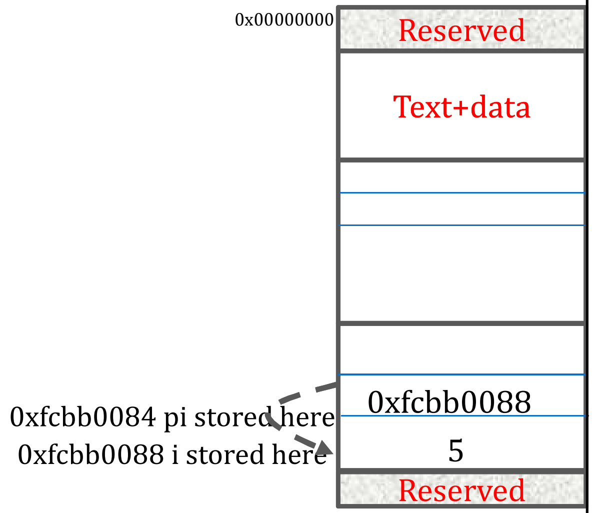
Example 2

```
main() {
    int i = 5;
    int *pi; //defines pointer to int

    pi = 0x124466c8; // correct syntax but where is that address

    *pi = 5678; // Expect blue screen of death or segmentation
fault. Don't do that
}
```



Example 2: Use of address of operator &

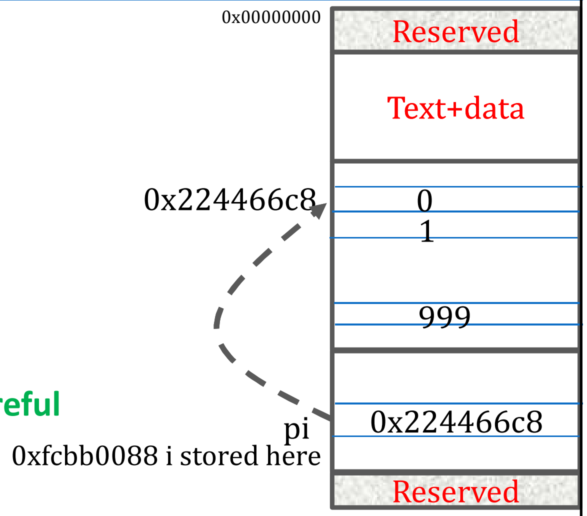
```
main() {
    int i = 5;
    int *pi; //defines pointer to int

    pi = &i; // let pi point to where i is stored

    *pi = 67;

     cout << i ; // what will be printed?

    i= 15;
    cout << *pi; // what will be printed?

    //Now delete pi it is not needed

    delete pi; // Unknown effect
    //→ Blue screen of death or segementation fault
     // Delete only to be used when pointer allocated with new
}
```



Pointers and Arrays

9

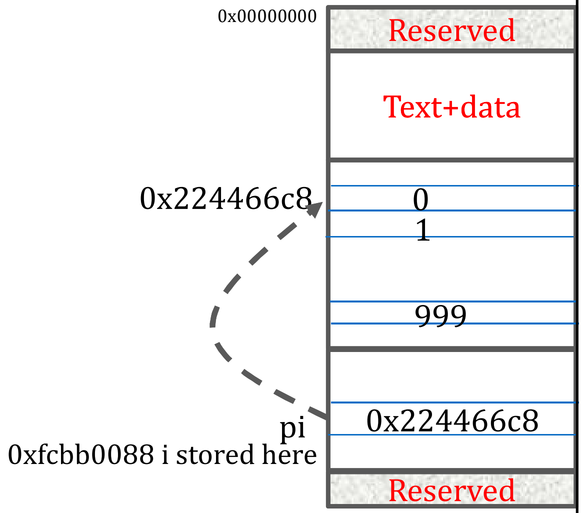
```
main() {
    int i ;
    int *pi; //defines pointer to int
    pi =  new int [1000]; // allocate array of 1000 ints to pi
    assert (pi !=NULL);
     for (i=0; i < 1000; i++)
          pi[i] = i;  // pi can be used as array
      //Alternatively
      for(i=0; i < 1000; i++)
                *(pi+i) = i * i;
     //OR
     for(i=0; i<1000; i++)
        *(pi++) = i*i; //but here pointer moves. Be careful

     delete [] pi; //de-allocate all elements
     // delete pi wil deallocate first one only
}
```



Pointers and Arrays

10

```
     delete [100] pi; //de-allocate first 100 elements

    pi =pi+100; //now point to 1st element in rest of 900 elements!
```



Pointers and Structures/Classes

11

```
main() {
   Complex x;
   Complex *cp; //defines pointer to Complex class

    cp =  new Complex(10.6,4.3);// allocate a complex ADT calling constructor 2
    assert (cp !=NULL);
   cp->ReadComplex(); //Call member function using arrow -> operator
   x.ReadComplex();
   cp->Add(x);
   //OR
   (*cp).Add(x);
   delete cp;

   cp = &x;// possible cp still alive
   cp - > Add(x); //No problem
}
```

pi = new Complex[2000];

Pointers and Structures/Classes

12

```
main() {
   struct Test{
        int i;
        char c;
    }
   Test *tp; //defines pointer to Test struct
    Test t;
    tp =  new Test;
    assert (to !=NULL);
   tp->i = 10//Call member function using arrow -> operator
   tp->c='v';

   *tp = t;
     delete tp;
}
```