Object Oriented Programming

Shaobai Kan

chapter 6

Outline

- Dynamic allocation and de-allocation of memory spaces

Dynamic allocation and de-allocation of memory spaces

Memory areas

Fact. Generally, programmers deal with five areas of memory

- Global name space
- The heap
- Registers
- Code space
- The stack

Memory areas (continue...)

- **Global name space.** Global variables are in global name space.

Memory areas (continue...)

- **Global name space.** Global variables are in global name space.
- **The stack.** Local variables are on the stack.

Memory areas (continue...)

- **Global name space.** Global variables are in global name space.
- **The stack.** Local variables are on the stack.
- **Code space.** Code is in code space.

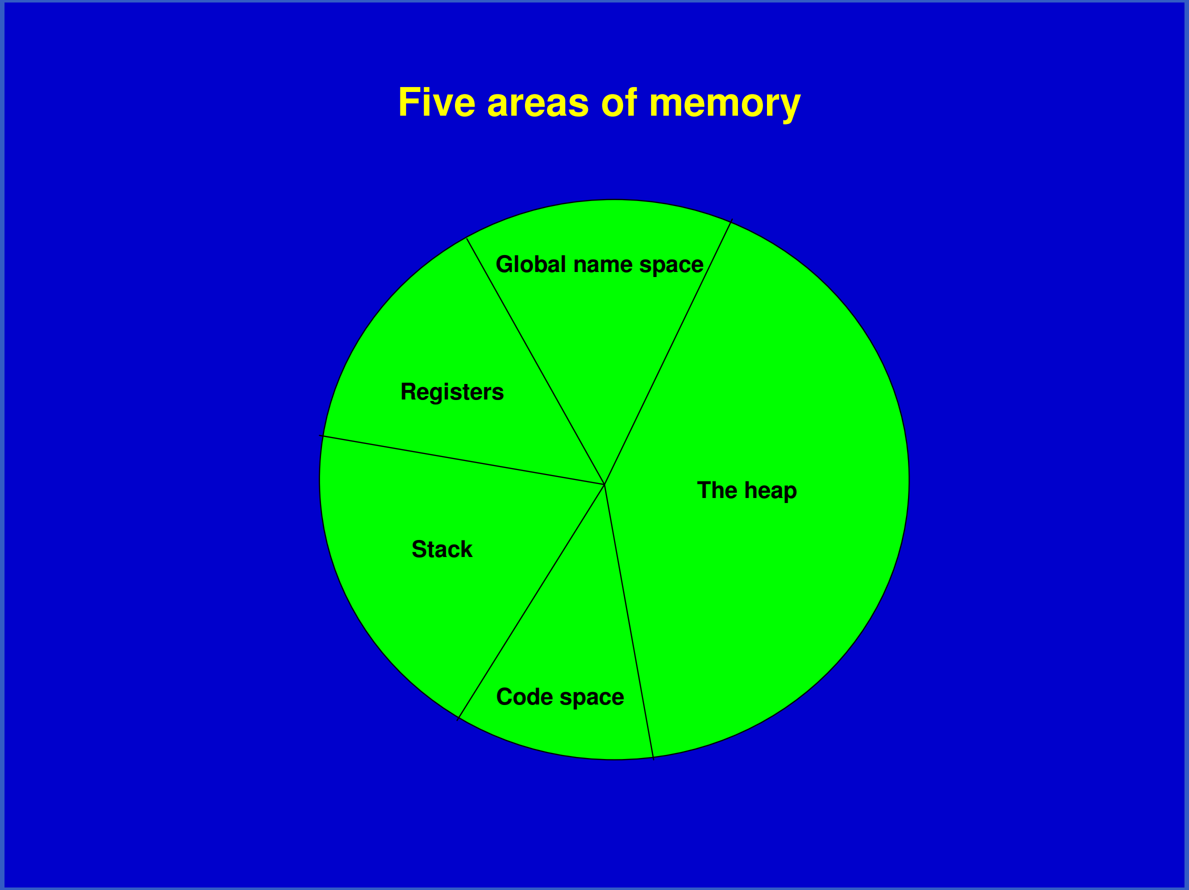
Memory areas (continue...)

- **Global name space.** Global variables are in global name space.
- **The stack.** Local variables are on the stack.
- **Code space.** Code is in code space.
- **Registers.** are used for internal housekeeping functions, such as keeping track of the top of the stack and instruction pointer.

Memory areas (continue...)

- **Global name space.** Global variables are in global name space.
- **The stack.** Local variables are on the stack.
- **Code space.** Code is in code space.
- **Registers.** are used for internal housekeeping functions, such as keeping track of the top of the stack and instruction pointer.
- **The heap.** About all remaining memory is given over to the heap; it is sometimes referred to as the free store.

Visual representation: memory areas



The heap

Q. Why should we bother to declare variables on the heap?

The heap

Q. Why should we bother to declare variables on the heap?

- Local variable. The problem with local variables is that they don’t persist; when the function returns, the local variables are thrown away.
- Global variable. Global variables solve that problem at the cost of unrestricted access throughout the program, which leads to the creation of code that is difficult to understand and maintain.

The heap

Q. Why should we bother to declare variables on the heap?

- Local variable. The problem with local variables is that they don’t persist; when the function returns, the local variables are thrown away.
- Global variable. Global variables solve that problem at the cost of unrestricted access throughout the program, which leads to the creation of code that is difficult to understand and maintain.

A: Putting data in the heap solves both of these problems.

The heap (Continue...)

Fact.

- The advantage to the heap is that the memory you reserve remains available until you explicitly free it; if you reserve memory on the heap while in a function, the memory is still available when the function returns.

The heap (Continue...)

Fact.

- The advantage to the heap is that the memory you reserve remains available until you explicitly free it; if you reserve memory on the heap while in a function, the memory is still available when the function returns.

- The advantage of accessing memory in this way, rather than using global variable, is that only functions with access to the pointer have access to data; it eliminates the problem of one function changing that data in unexpected and unanticipated ways.

The heap (Continue...)

Fact.

- The stack is cleaned automatically when a function returns; all local variables are removed from the stack.

The heap (Continue...)

Fact.

- The stack is cleaned automatically when a function returns; all local variables are removed from the stack.
- The heap is not cleaned until your program ends. It is your responsibility to free any memory that you've reserved when you are done with it; Otherwise, it will cause a memory leak.

Using the new operator

Fact. In C++, the memory on the heap can be allocated by using the new operator; new is followed by the type of the object that you want to allocate.

Using the new operator

Fact. In C++, the memory on the heap can be allocated by using the new operator; new is followed by the type of the object that you want to allocate.

Example 1.

```
unsigned short int * pPtr;
pPtr = new unsigned short int;
```

Using the new operator

Fact. In C++, the memory on the heap can be allocated by using the new operator; new is followed by the type of the object that you want to allocate.

Example 1.

```
unsigned short int * pPtr;
pPtr = new unsigned short int;
```

Example 2.

```
unsigned short int * pPtr = new unsigned short int;
```

Using the delete operator

Fact. When you are finished with your area of memory, you must call delete on the pointer; delete returns the memory to the heap.

Using the delete operator

Fact. When you are finished with your area of memory, you must call delete on the pointer; delete returns the memory to the heap.

Example 1.

```
unsigned short int * pPtr = new unsigned short int;
......
delete pPtr;
```

Using the delete operator

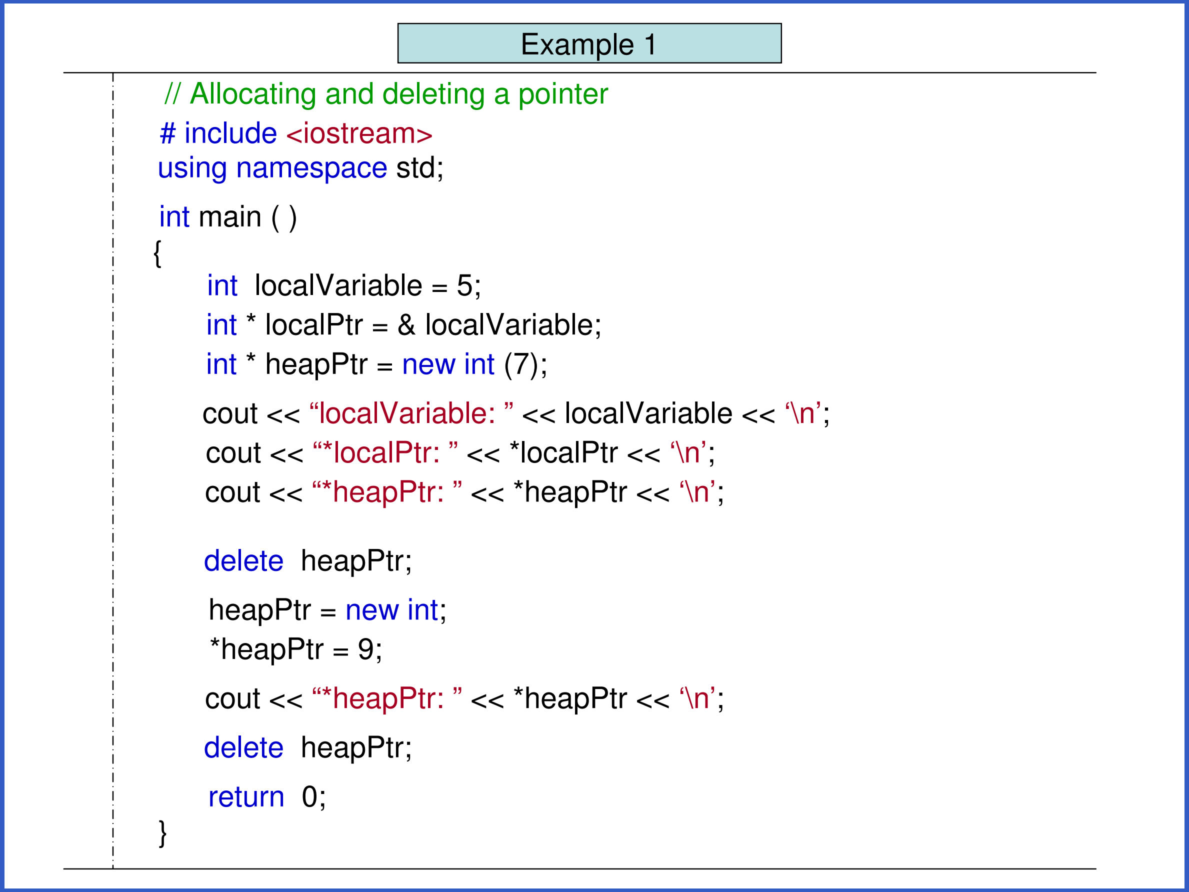
Fact. When you are finished with your area of memory, you must call delete on the pointer; delete returns the memory to the heap.

Example 1.

```
unsigned short int * pPtr = new unsigned short int;
                    ......
                 delete pPtr;
```

Fact. When you delete the pointer, what you are really doing is freeing up the memory whose address is stored in the pointer.

Operators: new and delete

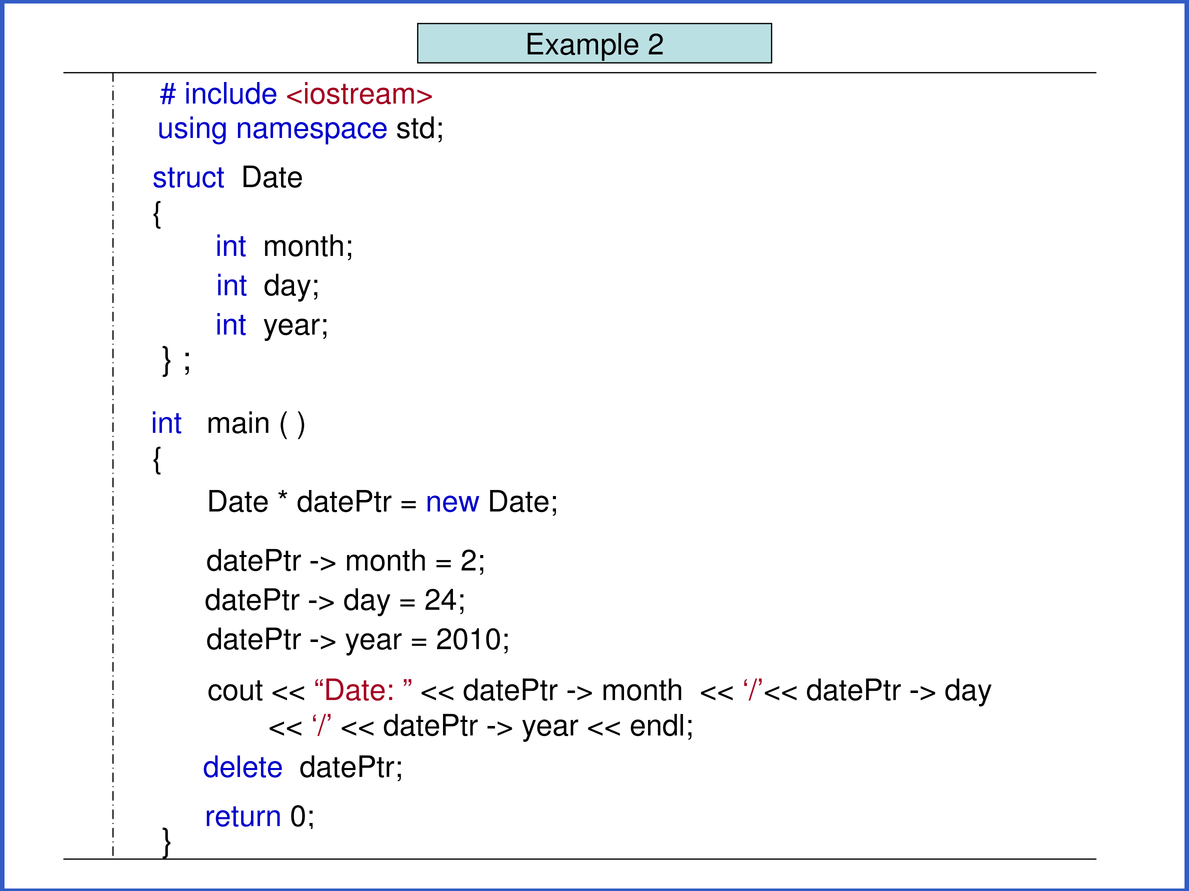


Example 1

```
// Allocating and deleting a pointer
# include <iostream>
using namespace std;
int main ( )
{
     int  localVariable = 5;
     int * localPtr = & localVariable;
     int * heapPtr = new int (7);
     cout << “localVariable: ” << localVariable << ‘\n’;
     cout << “*localPtr: ” << *localPtr << ‘\n’;
     cout << “*heapPtr: ” << *heapPtr << ‘\n’;

     delete  heapPtr;
     heapPtr = new int;
     *heapPtr = 9;
     cout << “*heapPtr: ” << *heapPtr << ‘\n’;
     delete  heapPtr;
     return  0;
}
```

Operators: new and delete (Continue...)



Example 2

```
# include <iostream>
using namespace std;

struct  Date
{
      int  month;
      int  day;
      int  year;
} ;

int   main ( )
{
      Date * datePtr = new Date;

      datePtr -> month = 2;
      datePtr -> day = 24;
      datePtr -> year = 2010;

      cout << “Date: ” << datePtr -> month  << ‘/’<< datePtr -> day
            << ‘/’ << datePtr -> year << endl;
      delete  datePtr;

      return 0;
}
```