C++

Memory Management

CO650 Advanced Programming

Topics

- Memory Management
  - The Stack
  - Memory
  - Destructors
  - Pointers
  - Auto Pointers
  - Memory Error Handler
- Performance
  - Inline functions
  - Register variables

Memory

A program utilises four types of memory

1. The code area : Compiled application resides
2. The global area : where global variables are located
3. The heap : dynamically allocated objects reside
4. The stack : parameters and local variables located

The Stack

- A data structure that restricts access. Last In First Out (LIFO).
  1. Add a new item
  2. Look at the last item added
  3. Remove the last item
- Statically declared variables are allocated memory from within the stack
- This memory is automatically freed when the variable goes out of scope.

The Stack

Here is the sequence of steps that takes place when a function is called:

1. The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
2. Room is made on the stack for the function's return type. This is just a placeholder for now.
3. The CPU jumps to the function's code. The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
4. All function arguments are placed on the stack.
5. The instructions inside of the function begin executing. Local variables are pushed onto the stack as they are defined.

```
int Add(int a, int b){
   int result = a + b;
   return result;
}
int main(){
      int temp;
      temp = Add(5,4);
      cout << temp;
}
```

Step	1
Adr cout	ox
temp	0

Step	2
return	
Adr cout	ox
temp	0

Step	3
Stack frame	
return	
Adr cout	ox
temp	0

Step	4
b	4
a	5
Stack frame	
return	
Adr cout	ox
temp	0

Step	5
result	
b	4
a	5
Stack frame	
return	
Adr cout	ox
temp	0

The Stack

When the function terminates, the following steps happen:

1. The function's return value is copied into the placeholder that was put on the stack for this purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.

```
int Add(int a, int b){
   int result = a + b;
   return result;
}
int main(){
      int temp;
      temp = Add(5,4);
      cout << temp;
}
```

Step	1
result	
b	4
a	5
Stack frame	
return	9
Adr cout	ox
temp	0

Step	2
Stack frame	
return	9
Adr cout	ox
temp	0

Step	3
Adr cout	ox
temp	9

Step	4
temp	9

Stack Overflow

- The stack has a limited size, and consequently can only hold a limited amount of information. If the program tries to put too much information on the stack, stack overflow will result. **Stack overflow** happens when all the memory in the stack has been allocated — in that case, further allocations begin overflowing into other sections of memory.
- Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) Overflowing the stack generally causes the program to crash.

Stack

The stack has advantages and disadvantages:

1. Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
2. All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
3. Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating large arrays, structures, and classes, as well as heavy recursion.

The Heap

- Dynamically variables are stored within the heap.
- Objects allocated on the heap can persist beyond the function in which they were instantiated. As long as a pointer is maintained.
- The heap is much larger than the stack and is typically used for storing objects and data structures.
- The dynamic memory is accessed through pointers or references.

New operator

- Allocates memory on the heap (free store)
- Returns the address of a chunck of memory

```
type *pointerName = new type
```

```
int *pAge = new int;
```

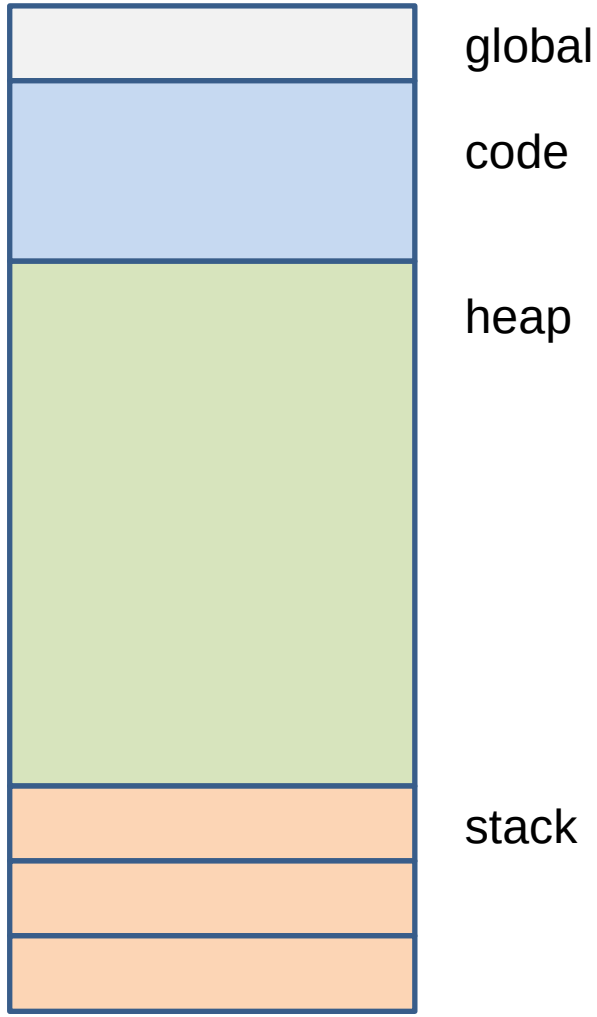
- The allocated memory can be initialised at the same time that it is allocated using a constructor

```
int *pAge = new int(21);
```

Memory Managment

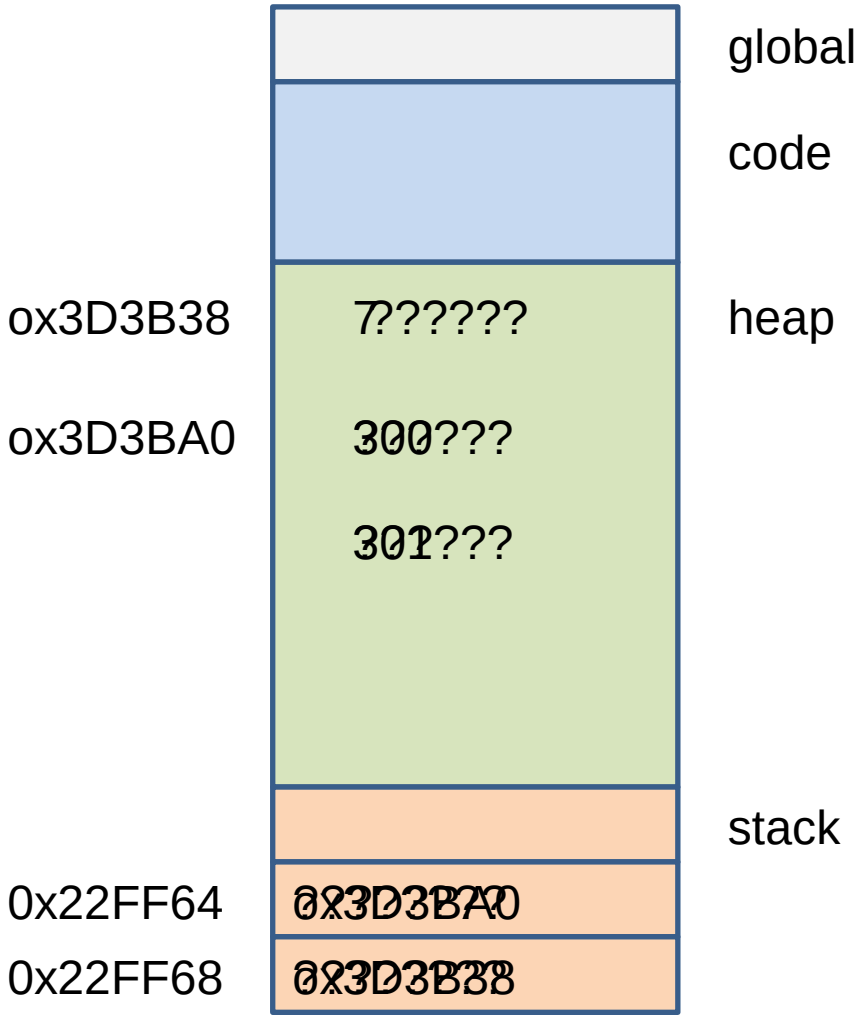
Four Main Memory Segments

1. Global : variables outside of stack & heap
2. Code: compiled program (read only)
3. Heap: dynamically allocated variables
4. Stack: parameters & temporary variables



Memory Managment

```
int * ptr;

ptr = new int;

*ptr = 7;

int *a;

a = new int[2];

*a = 300;
*(a+1) = 301;
```



Memory Managment

- Dynamic memory should be freed when the object is no longer required.
- Unlike variables stored on the stack heap memory must be explicitly freed.
- Delete must be followed by a pointer containing either a value of NULL or an area of memory allocated by the new keyword.

```
delete pointer;
```

```
MyClass *myObject = new MyClass();
// Process object myObject
delete myObject;
```

Delete arrays

- New used to allocate memory for an array

```
type  *pointer = new type[size];
```

- Array memory is deallocated using delete[]

```
delete[] pointer;
```

- Delete must be followed to a pointer containing either a value of NULL or an area of memory allocated by the new keyword.

```
int *marks = new int[100]
// Process marks
delete[] marks;
```

Delete Arrays of Pointers

```
class GameObject{}

GameObject ** objects = new GameObjects*[10];

for(int n = 0; n < 10; n++){
     objects[n] = new GameObject();
}

for(int n = 0; n < 10; n++){
     delete objects[n];
}
delete[ ] objects;
```

Pointer elements must have their memory freed before the array is deleted.

Dangling Pointers

- While delete frees the heap memory associated with a pointer, the pointer still points to this memory.
- This raises a problem as the computer may allocate this memory to another task.
- To avoid this problem either
  - assign 0 to the pointer.
  - Or assign it a valid memory address

```
int *pMark = new int(81);
// Process mark
delete pMark;
pMark = 0;
//…
If (pMark != 0) // Process pMark
```

Memory Leaks

- Caused by allocating memory and not freeing it.

```
void leak() {
    int* mark = new int(45);
}
```

- Mark is a local variable that goes out of scope when the function ends. The solution is to either free the memory within the function or return the pointer so that it can be freed in the calling code.
- Reassigning memory to a pointer can also produce leaks. While the memory storing the value 66 is freed that storing 45 is still allocated.

```
void leak() {
    int* mark = new int(45);
     mark = new int(66);
    delete mark;
}
```

Destructors

- Is invoked automatically just prior to object destruction
- Used to release dynamically allocated memory and free resources
- Has the same name as the Class but is prefixed with the tilde character ~
- Must be declared public
- Takes no arguments
- Has no return type
- If your class maintains memory on the heap, this memory should be freed within the destructor.

Destructors Example

```
class Game{
private:
        NPC *npcs;
public:
        Game(int numberNPC){
                npcs = new NPC[numberNPC];
        }
        ~Game(){
                delete[ ] npcs;
        }
};
```

Detecting Memory Error

- If new fails to allocate memory then it will return a zero or NULL pointer
- This will occur when there is insufficient memory
- Use this to capture memory errors

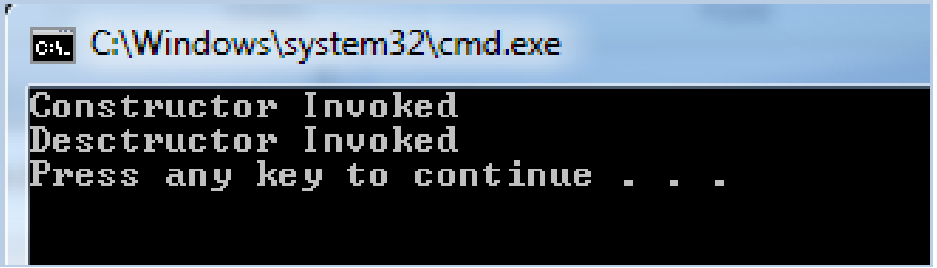
```
GameObject* objArray = NULL;
objArray = new GameObject[size];
if(objArray== NULL){
        // Handle error
}
```

Auto Pointers

- Performs automatic cleanup on dynamically allocated objects when no longer needed.

```
auto_ptr<type-id>identifier(new type-id);
```

```
class GameObject{
public:
    GameObject(){ cout << "Constructor Invoked" << endl; }
    ~GameObject(){ cout << "Desctructor Invoked" << endl;}
};

void f(){
      auto_ptr<GameObject> obj(new GameObject());
}

int main(){
      f();
}
```



Auto Pointers

- In the previous example obj is a local pointer that is allocated memory dynamically. Normally the destructor would only be invoked by an explicit delete command.
- As it is an auto_ptr it deletes itself when no longer required.

Handling Memory Errors

- New_handler is an internal C++ pointer that points to the function that will be invoked when a memory allocation fails.
- Developers can define an error handling function to handle errors and assign it to new_handler by passing the function name to the set_new_handler function

```
set_new_handler(ErrorHandlerfunctionName);
```

- set_new_handler is defined within <**new.h>**
- The Error Handler Function should return void and have no parameters

```
void MemoryErrorHandler(){
        cerr << "You have run out of memory! \n";
        exit(1);
}
.....
set_new_handler(MemoryErrorHandler);
```

inline Functions

- The function is expanded during compilation, where ever the function is invoked. The code within the function replaces the invocation.
- Typically used for small functions.
- There is no guarantee that the function will be expanded, it is just a recommendation to the compiler.
- A complex or recursive functions is not a suitable candidate for inlining.
- A function defined within the body of the class is an inline function.
- Alternatively if the function's declaration includes the inline keyword then the definition will be inline.
- If the definition includes the inline keyword and the declaration doesn't the function will be inline.

Three inline Functions

```
class ClassName{
    type Identifier1(pramater/s){
        statment/s
    }
    inline  type Identifier2(pramater/s);
    returnType Identifier3(pramater/s);
}

type ClassName::Identifier2(parameter/s){
}

inline type ClassName::Identifier3(parameter/s){
}
```

inline example

```
class Fast{
private:
        int data;
public:
        Fast(){
                data = 0;
        }
        void SetData(int value){
                data = value;
        }
        inline bool IsEven();
        void Increment();
};

bool Fast::IsEven(){
        return (data % 2 == 0);
}

inline void Fast::Increment(){
        data++;
}
```

Register Variables

- Can possibly improve the performance of certain operations.
- By labelling a variable with the register keyword you are recommending to the compiler that it is placed in the register.
- Only local variables or formal parameters (not global or static).

```
register type identifier;
```

- There was no appreciable difference in performance in using a register variable in the example below. Compiler optimization negates the use of register.

```
for(register int n =0 ; n < 100000000; n++)
        fast.Increment();
```

Summary

- C++ doesn’t have garbage collection, so it is the programmers responsibility to manage resources.
- Various mechanisms can be used to enhance the programs performance. However compiler optimisation may negate their use.

References

- http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/