CO650 Advanced Programming
Stack, Heap, Memory Management & Destructors

Memory Management

- The Stack
- Memory
- Destructors
- Pointers
- Auto Pointers
- Memory Error Handler

Performance
- Inline functions
- Register variables

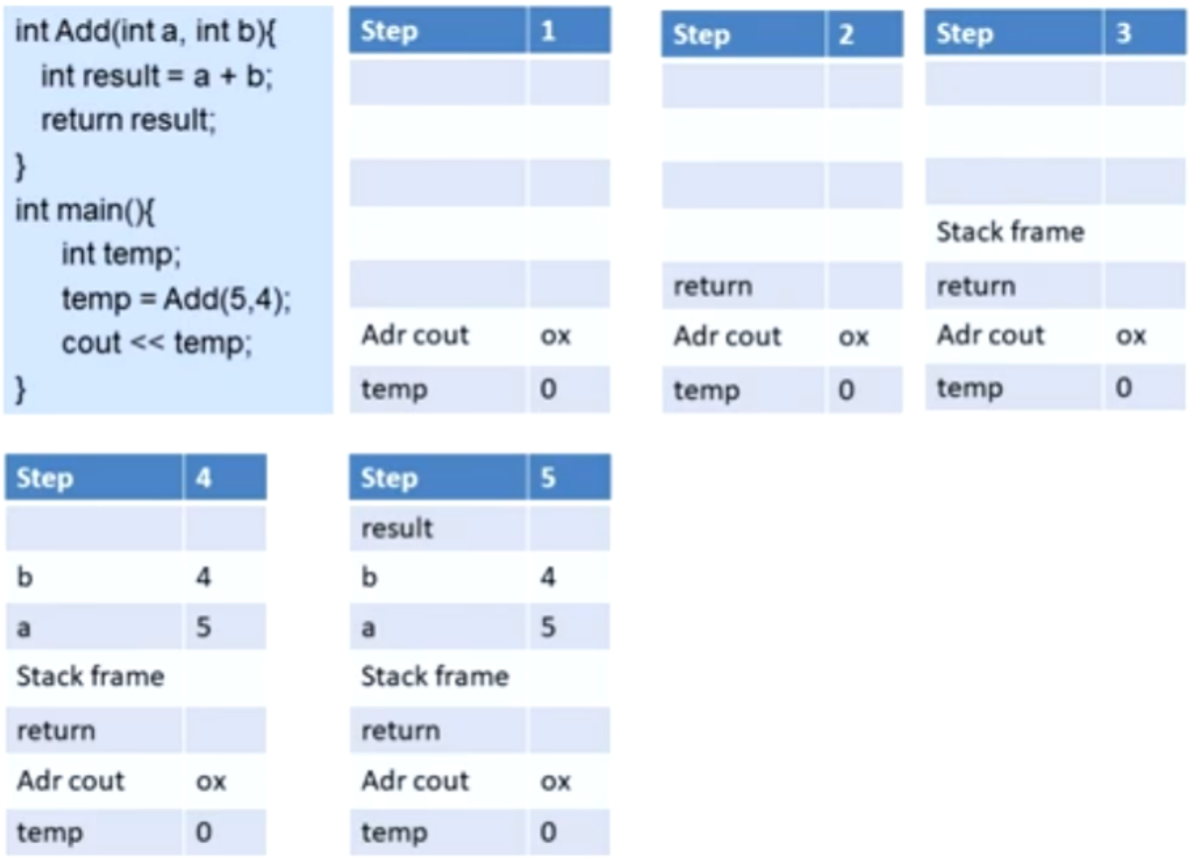
Memory

A program utilizes four types of memory:

1. The code area: compiled application
2. The global area: global variables
3. The heap: dynamically allocated objects
4. The stack: parameters and local variables \

The Stack

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



Example of adding items to the stack in order.

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

Process of removing stacks from previous example.

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, B calls C, C calls D, etc…) Overflowing the stack generally causes the program to crash

Advantages of Stack

1. Stack automatically manages the memory and garbage collection. 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.

.

The "New" Operator

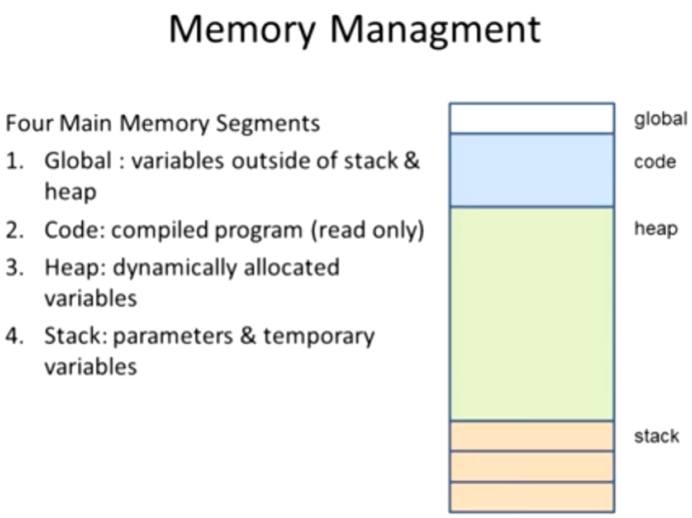
The new operator allocates memory on the heap and returns the address of a chunk of memory.

```
type *pointerName = new type;
int *pAge = new int;
```

The allocated memory can be initialized at the same time that it is allocated using a constructor.

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

Memory Management



- 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.

Deleting Object allocated on the Heap

```
delete pointer
```

Example:

```
MyClass *myObject = new MyClass();

// Process object myObject

delete myObject;
```

Allocating Arrays on the Heap

- New used to allocate memory for an array

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

- Array memory is deallocated using delete[]. Without [] only the address of the first element will be addressed.

```
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;
```

Deleting Array of Pointers

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

```
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;
```

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

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.
- Reasoning memory to a pointer can also produce leaks. While the memory storing the value 66 is freed, memory storing 45 is still allocated.

```
void leak() {
```

```
int*mark = new int(45);
mark = new int(66);
delete mark;

}
```

Destructors

Destructor is invoked automatically just prior to object destruction. It is used to release dynamically allocated memory and free resources.It has the same name as the Class but is prefixed with the tilde character ~. It must be declared public, takes no arguments, and has no return type. If your class maintains memory on the heap, this memory should be freed within the destructor.

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 eros

```
GameObject* objArray = NULL;
objArray = new GameObject[size];

if(objArray==NULL){

// Handle Error

}
```

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 eros 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

Example:

```
void MemoryErrorHandler() {

cerr << “You have run out of memory! \n”;
exit(1);

}

set_new_handler(MemoryErrorHandler);
```

Auto Pointers

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

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

```
class GameObject{
```

```
void f() {

auto_ptr<GameObject> obj(new GameObject());

}

int main() {

f();

}

}
```

In this example, auto pointer frees up memory on the heap automatically once function f terminates.

Memory Optimization

Inline Functions

- The function is expanded during compilation, wherever 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 function 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.

Examples (in pseudocode) :

```
class ClassName {

type Identifier1(parameters) {

statements

}

inline type Identifier2(parameters);
```

returnType Identifier3(parameters);

}

type ClassName::Identifier2(parameters){ }

inline type ClassName::Identifier3(parameters){ }

Example in full code:

```
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

- Register variables may improve the performance of certain operations.
- By labeling 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) can be used and it allows fast read and write access.
- We do not get access to the memory address of items declared on the register.

Side Note

Register method may have been more noticeable back in time where there were lower specifications of machines. However recent tests have shown hardly noticeable improvement. Inline functions are still relevant and still have potential to be useful