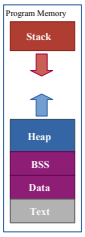
ADVANCED PROGRAMMING IN C++

Dynamic Memory

Patrick Bader · SS 2023

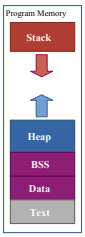
Local variables

- Live until their scopes end, i.e. on closing curly brace: }
- Allocated on the **Stack**.
- Objects are automatically destroyed.



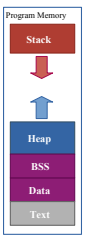
Global variables

- Live as long as the program runs.
- Allocated in **Text**, **Data**, or **BSS** segment of program.
- Objects are automatically destroyed.



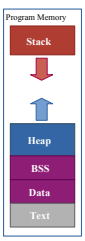
Why Dynamic Memory Allocation?

- How to refer to the same object from different locations?
- How to implement a class with storage requirements defined at runtime, e.g. std::vector?
- What about polymorphic usages of classes?



Solution

- Allocate chunks of memory at runtime.
- Use a separate program segment, called **Heap**.
- Manually manage free and allocated memory in heap.
- Refer to memory on heap using **pointers**.



Pointers

Similar to references.

Contain addresses of artibtrary memory locations.

Memory addresses are just numbers with a fixed size.

Target addresses may be changed (pointer arithmetic).

May refer to no value at all.

Pointers - Syntax

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};

// Declare a variable x which is a pointer to an integer value
int* ptr;


ptr = &x;


std::cout << *ptr << "\n";
```

Pointers - Syntax

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};

// Declare a variable x which is a pointer to an integer value
int* ptr;

//    Take the memory address of variable x and store this address in the ptr variable
ptr = &x;


std::cout << *ptr << "\n";
```

Pointers - Syntax

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};

// Declare a variable x which is a pointer to an integer value
int* ptr;

//    Take the memory address of variable x and store this address in the ptr variable
ptr = &x;

//           Get a reference to the value at the memory address stored in ptr
std::cout << *ptr << "\n";
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

// -->
ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	1
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	0x????

Output:

```
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
// -->
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	1
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	**0x1020**

Output:

```
10
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
// -->
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	1
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	**0x1024**

Output:

```
10
20
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
// -->
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	1
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	**0x1028**

Output:

```
10
20
1
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
// -->
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	1
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	0x1028

Output:

```
10
20
1
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
// -->
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	**13**
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	0x1028

Output:

```
10
20
1
13
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
// -->
ptr = ptr - 1;
std::cout << *ptr << "\n";
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	13
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	**0x102C**

Output:

```
10
20
1
13
2
```

Pointers - Example

```
int x = 10;
int y = 20;
int a[] = {1, 2, 3};
int* ptr;

ptr = &x;
std::cout << *ptr << "\n";
ptr = &y;
std::cout << *ptr << "\n";
ptr = a;
std::cout << *ptr << "\n";
ptr = &a[0];
*ptr = 13;
std::cout << *ptr << "\n";
++ptr;
std::cout << *ptr << "\n";
ptr = ptr - 1;
std::cout << *ptr << "\n";
// -->
```

Address	Type	Name	Value
0x1020	int	x	10
0x1024	int	y	20
0x1028	int	a[0]	13
0x102C	int	a[1]	2
0x1030	int	a[2]	3
0x1034	int*	ptr	**0x1028**

Output:

```
10
20
1
13
2
13
```

Pointers - Null Pointers

C++11 defines a special type for null pointers which is called `nullptr_t`

There is a single instance of this type, called `nullptr`.

Converts implicitly to any pointer type with address `0x0000` and to `false`.

No pointer arithmetic allowed on `nullptr_t`.

`nullptr_t` can be used for overloading.

In old or C code, you will find 0 or NULL for null pointers. Do not use these anymore!

Pointers - Null Pointers

```
void print(char* text) {
if(text != nullptr)
    std::cout << "Text: " << text << "\r\n";
}
void print(int number) {
    std::cout << "Number: " << number << "\r\n";
}
int main(int argc, char* argv[]){
    print("test"); // prints: "Text: test"
    print(3);
    // prints: "Number: 3"
    print(0);
    // prints: "Number: 0"
    print(NULL);
    // prints: "Number: 0"
    print(nullptr);
    // calls print function with char* parameter and prints nothing
    return 0;
}
```

One Of The Hardest Programming Problems...

One Of The Hardest Programming Problems...

... in two lines of code:

One Of The Hardest Programming Problems...

... in two lines of code:

```
#include <cstdlib>

void* ptr = malloc(16); // allocate 16 bytes of memory from heap

free(ptr); // Free up the previously allocated memory
```

One Of The Hardest Programming Problems...

... in two lines of code:

```
#include <cstdio>

File* file = fopen("myfile.txt", "w"); // open a file

fclose(file); // close the file
```

One Of The Hardest Programming Problems...

... in two lines of code:

```
#include <cstdlib>

void* ptr = malloc(16);

free(ptr);
```

One Of The Hardest Programming Problems...

... in two lines of code:

```
#include <cstdlib>

void* ptr = malloc(16);
func();
free(ptr);
```

One Of The Hardest Programming Problems...

... in two lines of code:

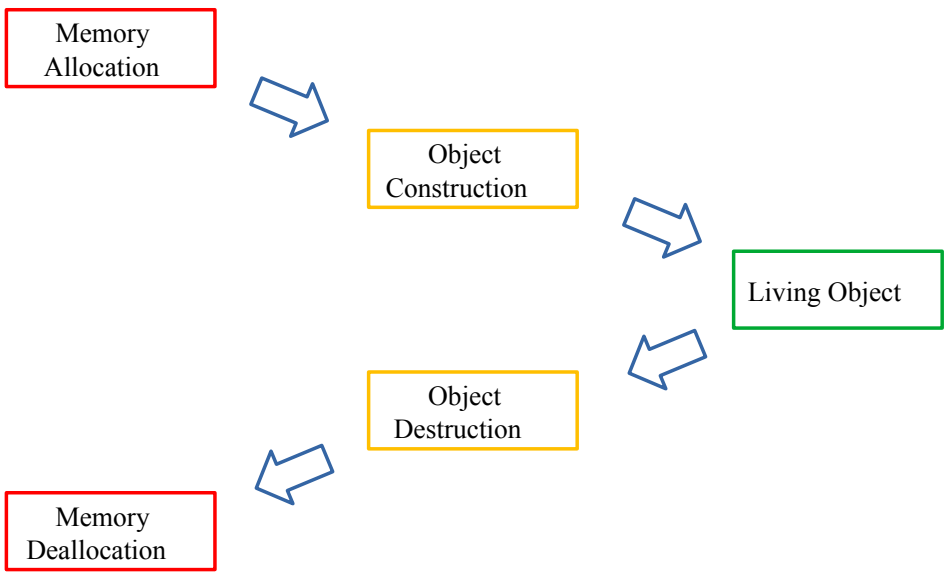
```
#include <cstdlib>

void func() { throw; }

void* ptr = malloc(16);

func();

free(ptr);
```

One Of The Hardest Programming Problems...

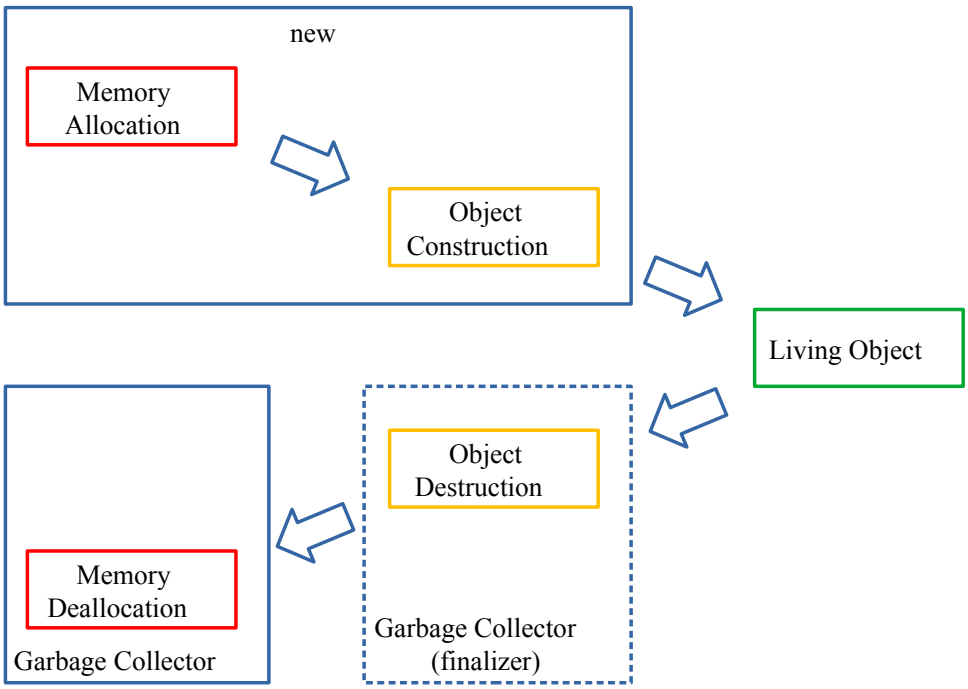
... in two lines of code:

```
#include <cstdlib>

void func() { throw; }

void* ptr = malloc(16);

try {
    func();
}
catch(...) {
    free(ptr);
    throw;
}

free(ptr);
```

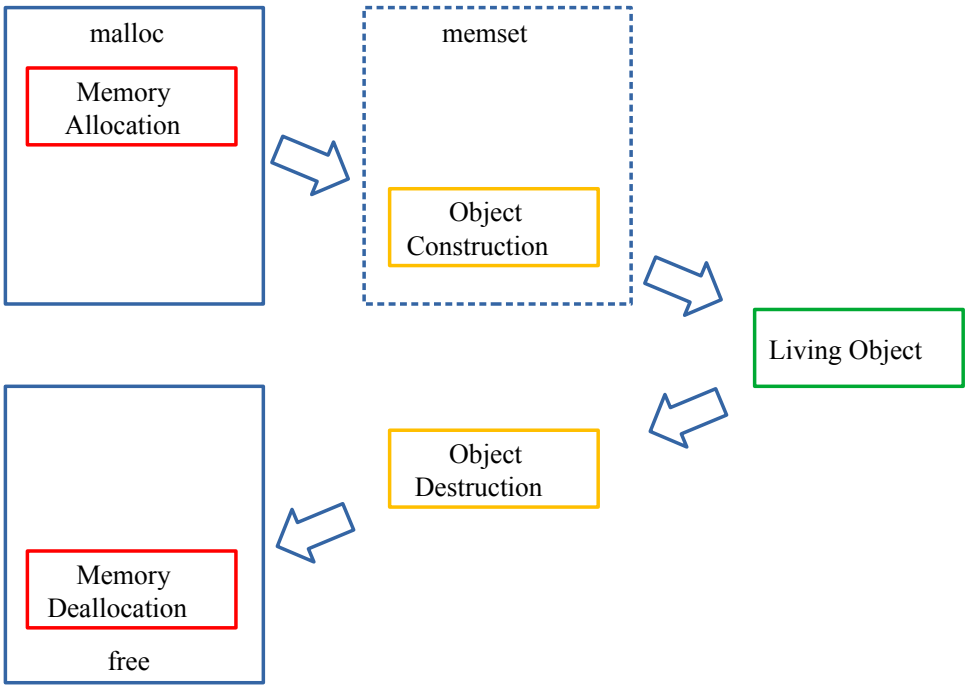
Object Lifecycle



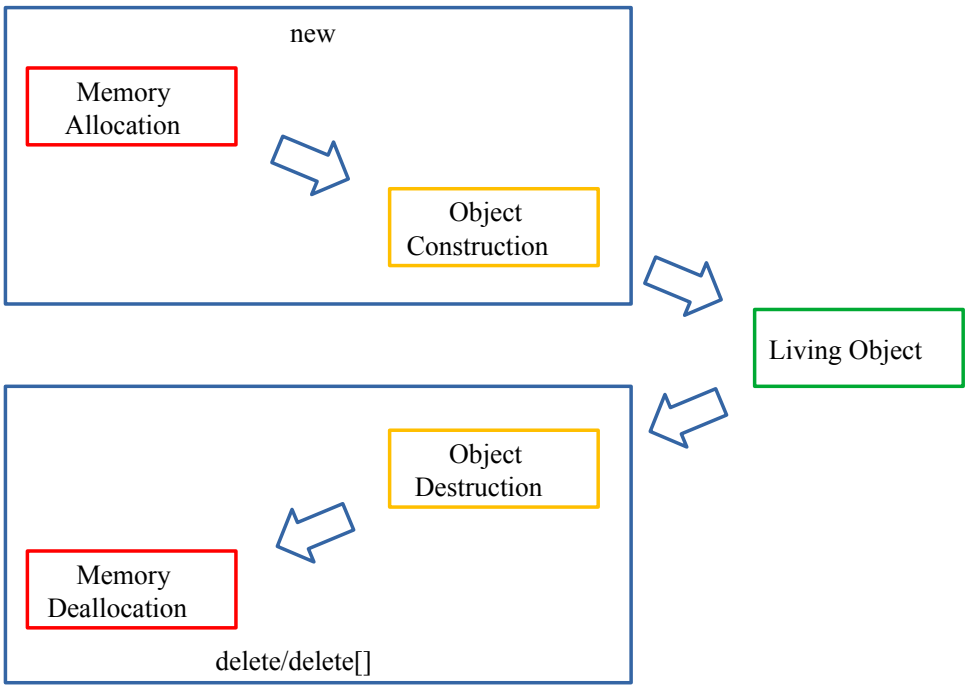
Object Lifecycle - Java



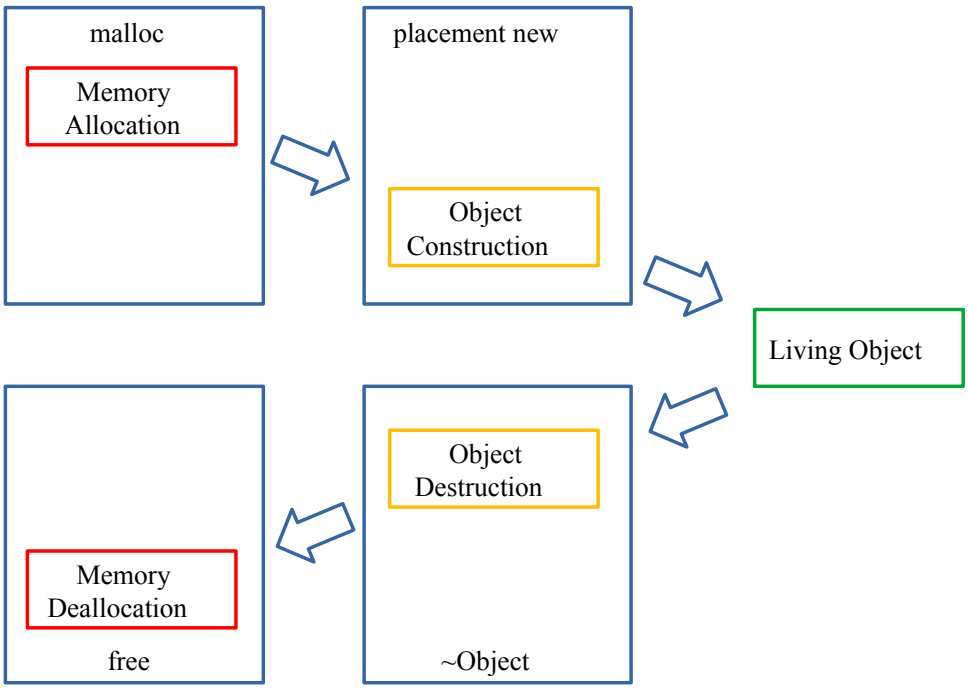
Object Lifecycle - C



Object Lifecycle - C++



Object Lifecycle - C++



One Of The Hardest Programming Problems...

... in two lines of code:

```
class Object {
};


Object* obj = new Object();


delete obj;
```

```


Object* obj = new Object[10];


delete[] obj;
```

There are **two** `delete` operators which must never be mixed:

- **`delete`** is for single objects allocated with **`new`**
- **`delete[]`** is for arrays allocated with **`new`** `Object[size]`