CS 225

Lab Debug

Learning Objectives

1. Apply the Debugging Process
2. Practice using pointers

Debugging Process

1. Understand the goal of the code

Debugging Process

1. Understand the goal of the code
2. Edge cases

Debugging Process

1. Understand the goal of the code
2. Edge cases
3. Does my code do what I want?

main.cpp

```cpp
#include <iostream>
#include "Cube.h"
using cs225::Cube;

int main() {
  Cube c;
  std::cout << "Address storing `c`:" << &c << std::endl;

  Cube *ptr = &c;
  std::cout << "Contents of ptr: "<< ptr << std::endl;
  std::cout << "Addr. storing ptr: "<< &ptr << std::endl;

  return 0;
}
```

What is the output of these lines?

Address storing `c`: 0x76b2f4d6c9a4

main.cpp

```
#include <iostream>
#include "Cube.h"
using cs225::Cube;

int main() {
  Cube c;
  std::cout << "Address storing `c`:" << &c << std::endl;

  Cube *ptr = &c;
  std::cout << "Contents of ptr: "<< ptr << std::endl;
  std::cout << "Addr. storing ptr: "<< &ptr << std::endl;

  return 0;
}
```

Address storing `c`: 0x76b2f4d6c9a4
Contents of ptr : 0x76b2f4d6c9a4 (same address as &c)
Addr. storing ptr: 0x7ffc2c6f9870 (different address than &c)

Pointers and References

A variable containing an instance of an object:

```
Cube s1;
```

A reference variable of a Cube object:

```
Cube & s1;
```

A variable containing a pointer to a Cube object:

```
Cube * s1;
```

main.cpp

```cpp
int main() {
  int  a = 225;
  int *c = &a;

  std::cout << "a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << "c = " << c << std::endl;
  std::cout << "&c = " << &c << std::endl;
  std::cout << "*c = " << *c << std::endl;

  return 0;
}
```

```
 a = 225
&a = 0x7ffeea4c23d8
 c =
&c =
*c =
```

main.cpp

```
int main() {
  int  a = 225;
  int *c = &a;

  std::cout << "a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << "c = " << c << std::endl;
  std::cout << "&c = " << &c << std::endl;
  std::cout << "*c = " << *c << std::endl;

  return 0;
}
```

```
 a = 225
&a = 0x7ffeea4c23d8
 c = 0x7ffeea4c23d8
&c =
*c =
```

main.cpp

```
int main() {
  int  a = 225;
  int *c = &a;

  std::cout << "a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << "c = " << c << std::endl;
  std::cout << "&c = " << &c << std::endl;
  std::cout << "*c = " << *c << std::endl;

  return 0;
}
```

```
 a = 225
&a = 0x7ffeea4c23d8
 c = 0x7ffeea4c23d8
&c = 0x7ffeea4c23d0
*c =
```

main.cpp

```
int main() {
  int  a = 225;
  int *c = &a;

  std::cout << "a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << "c = " << c << std::endl;
  std::cout << "&c = " << &c << std::endl;
  std::cout << "*c = " << *c << std::endl;

  return 0;
}
```

```
 a = 225
&a = 0x7ffeea4c23d8
 c = 0x7ffeea4c23d8
&c = 0x7ffeea4c23d0
*c = 225
```

The (->) arrow operator

```
int main() {
  Cube c;
  Cube *ptr = &c;
  c.w=12;
  (*ptr).w = 12;
  ptr->w = 12;
  std::cout << ptr->w;


  return 0;
}
```

Indirection Operators

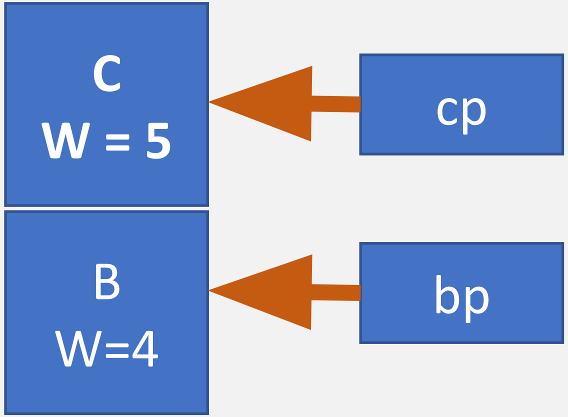
&c – return memory address storing the variable

*ptr – return the content pointed by ptr

ptr-> access members of a class through a pointer

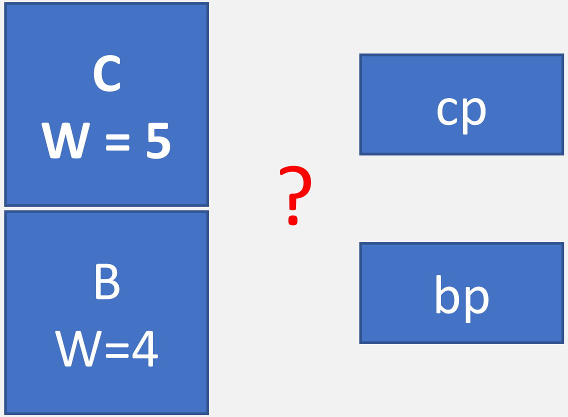
Manipulating pointers:

```
int main() {
  Cube c(5);
  Cube b(4);

  Cube *cp = &c;
  Cube *bp = &b;



  return 0;
}
```



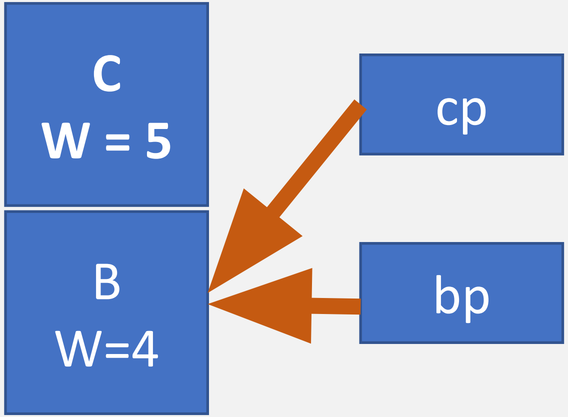
Manipulating pointers:

```
int main() {
  Cube c(5);
  Cube b(4);

  Cube *cp = &c;
  Cube *bp = &b;

  cp = bp

  return 0;
}
```



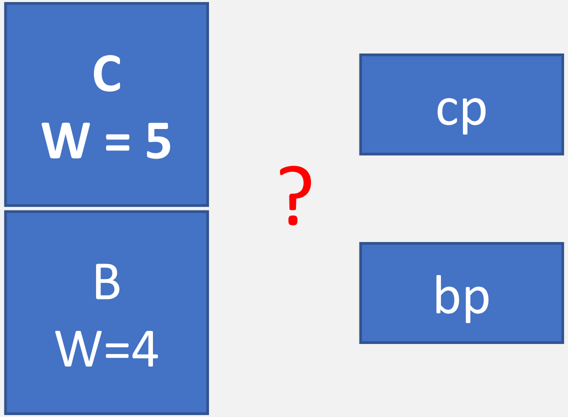
Manipulating pointers:

```
int main() {
   Cube c(5);
   Cube b(4);

   Cube *cp = &c;
   Cube *bp = &b;

   cp = bp

   return 0;
}
```



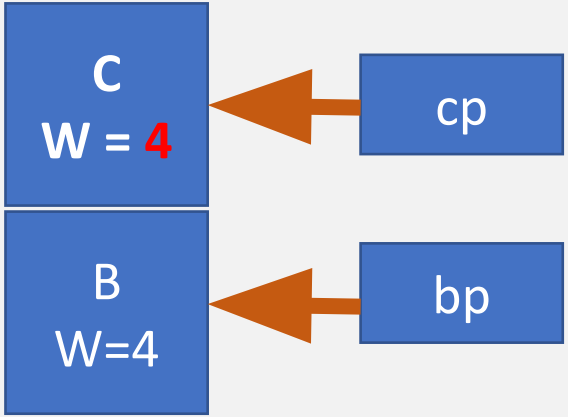
Manipulating pointers:

```
int main() {
  Cube c(5);
  Cube b(4);

  Cube *cp = &c;
  Cube *bp = &b;

  *cp = *bp

  return 0;
}
```



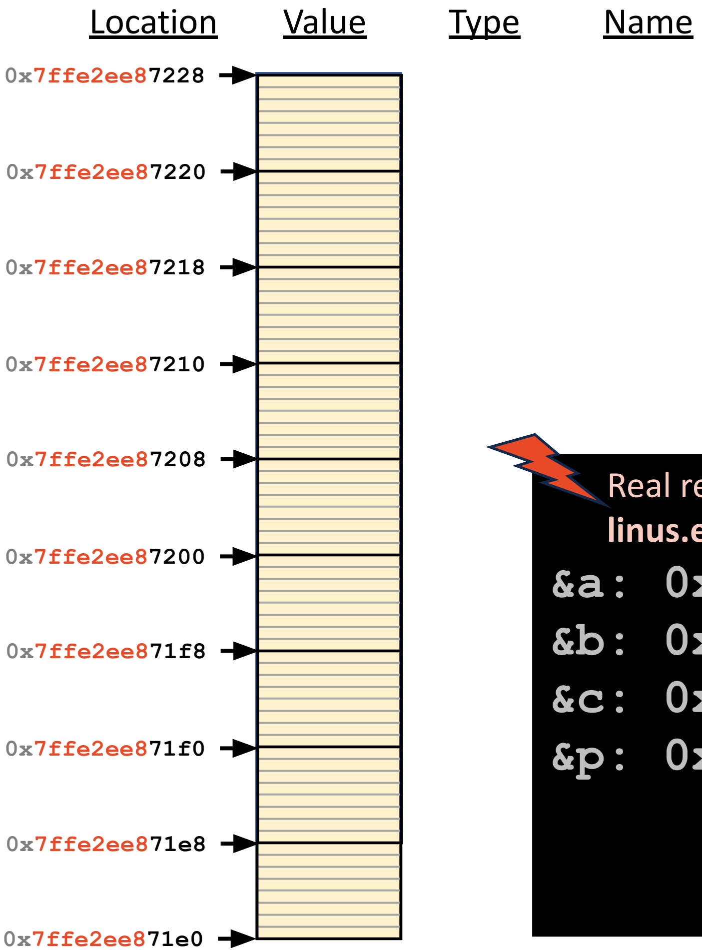
Manipulating pointers:

```
int main() {
   Cube c(5);
   Cube b(4);

   Cube *cp = &c;
   Cube *bp = &b;

   *cp = *bp

   return 0;
}
```



Stack and Heap

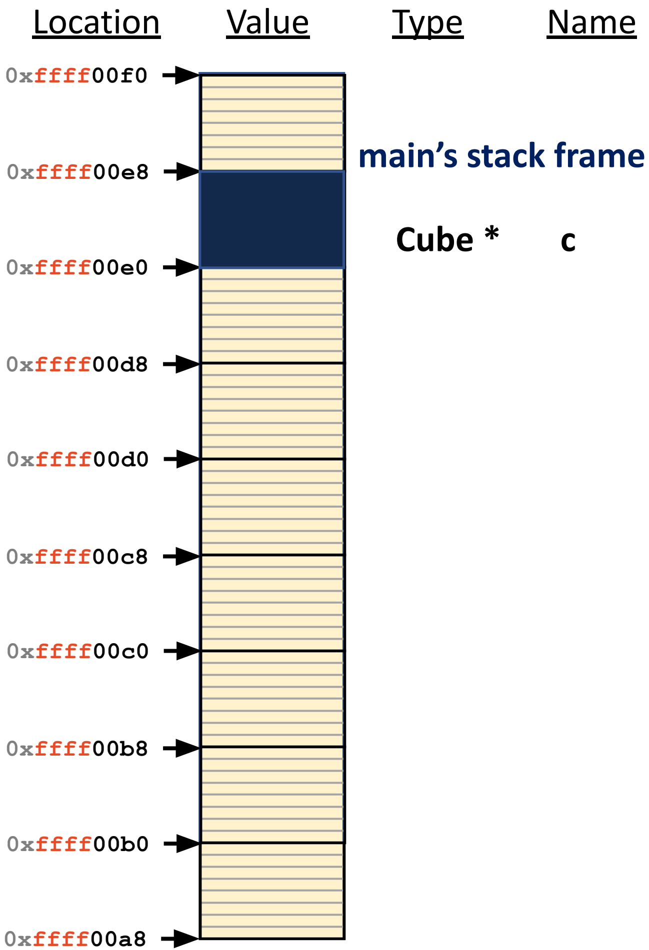
example1.cpp




```cpp
int main() {
  int a;
  int b = -3;
  int c = 12345;

  int *p = &b;

  return 0;
}
```

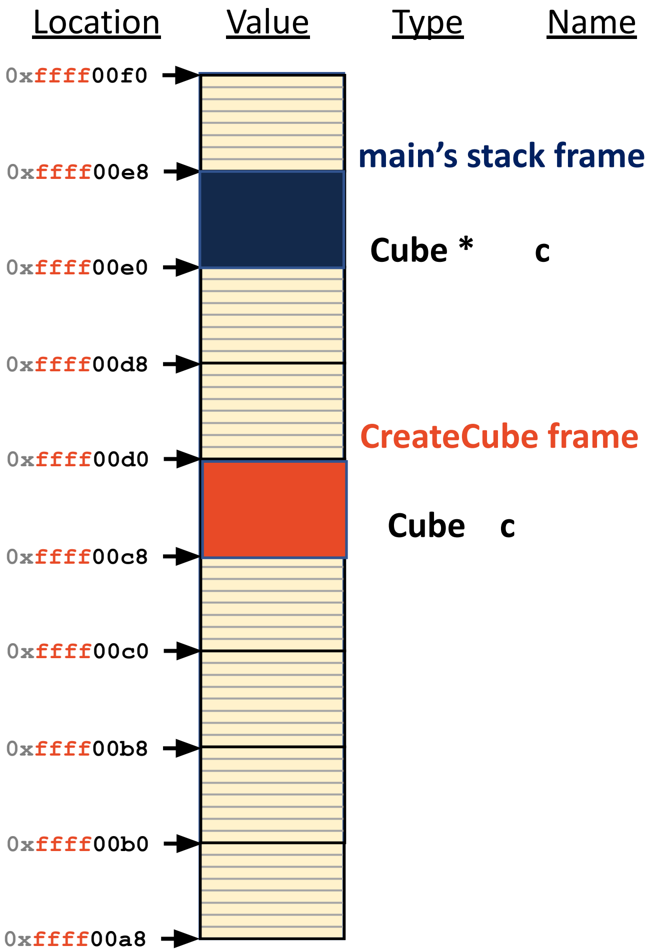
Real results when running on **linus.ews.illinois.edu**

```
&a:  0x7ffe2ee87218
&b:  0x7ffe2ee87214
&c:  0x7ffe2ee87210
&p:  0x7ffe2ee87208
```



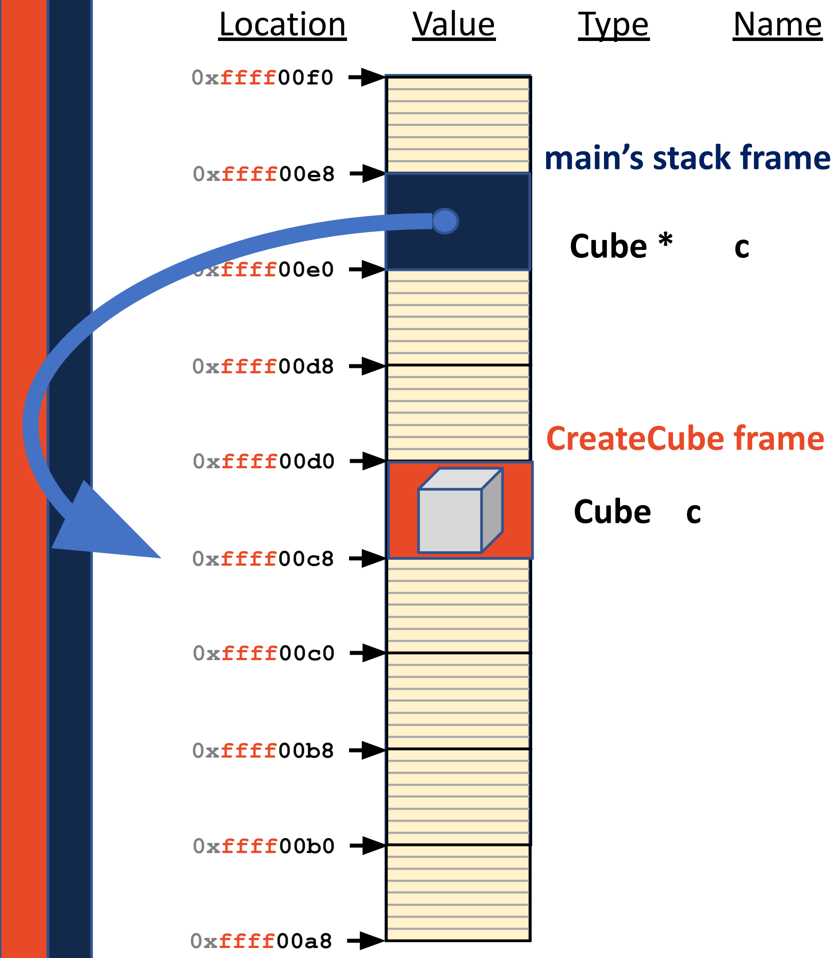
puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```



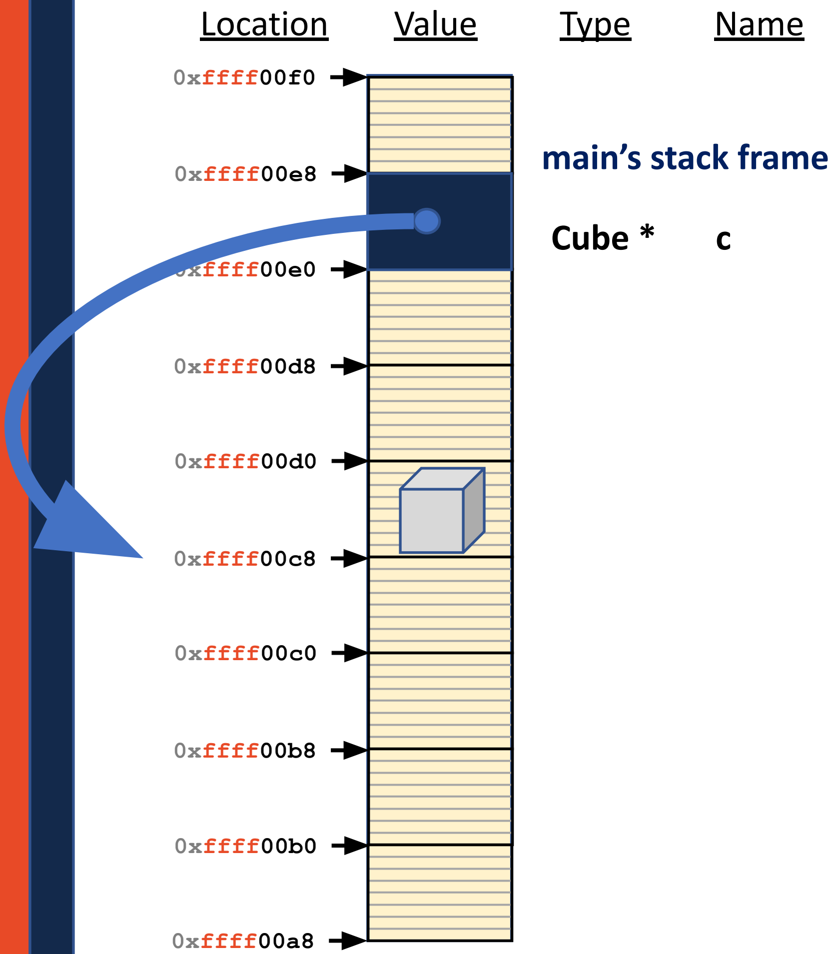
puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube* CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```



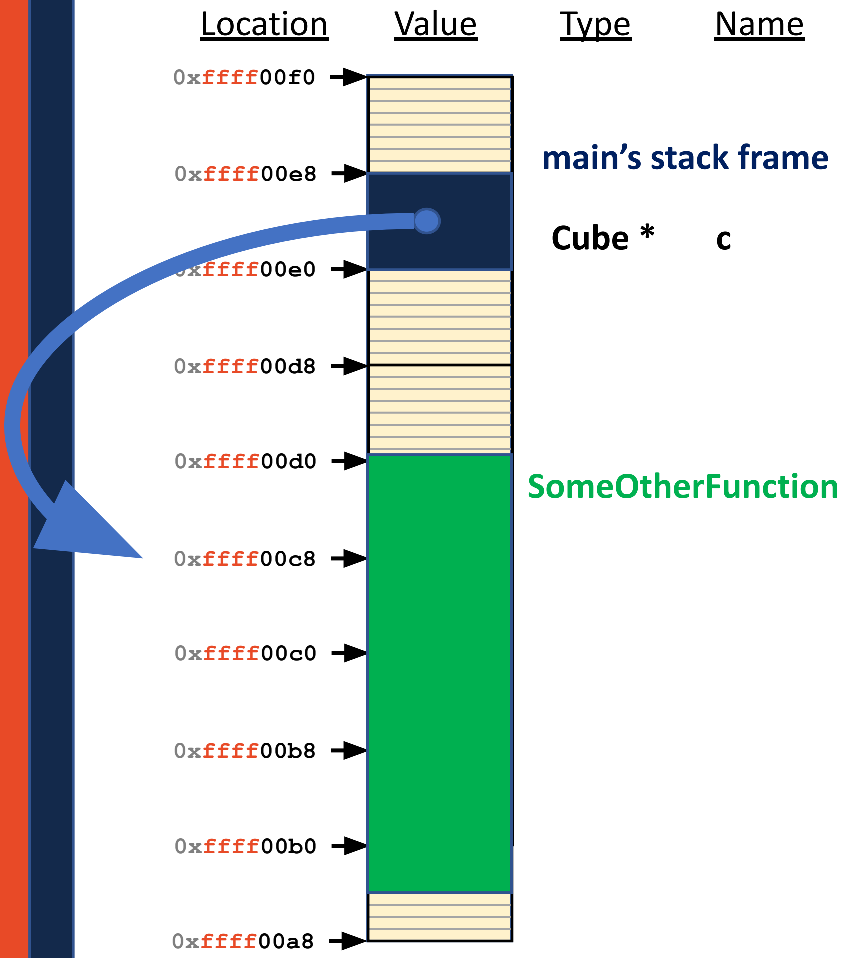
puzzle.cpp

```cpp
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```



puzzle.cpp

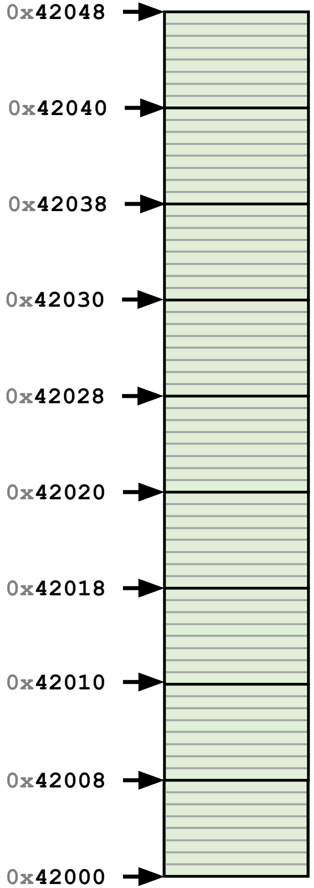
```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

Heap Memory



The only way to create heap memory is with the use of the **new** keyword. Using **new** will:

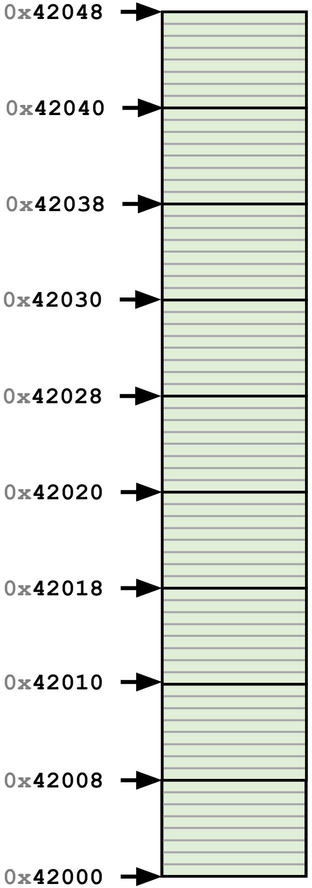
1.

2.

3.

```
Cube *c = new Cube();
```

Heap Memory



The only way to create heap memory is with the use of the **new** keyword. Using **new** will:

1. **Allocate memory**
2. **Construct an object**
3. **Return the pointer**

```
Cube *c = new Cube();
```

Heap Memory - delete

2. The <u>only</u> way to free heap memory is with the use of the **delete** keyword. Using **delete** will:

-
-

3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be "leaked memory".

Heap Memory - delete

2. The <u>only</u> way to free heap memory is with the use of the **delete** keyword. Using **delete** will:

- Destruct the object
- Release memory back to the system

3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be “leaked memory”.

Segmentation Fault

A segmentation fault is when your program attempts to access memory it has either not been assigned by the operating system, or is otherwise not allowed to access.

```
int main() {
  Cube *c = NULL;
  std::cout<<c->width;

  return 0;
}

```

```
int main() {
  Cube *c = new Cube();
  std::cout<<c->width;

  Cube *b = c;
  delete c;
  delete b;
  return 0;
}
```