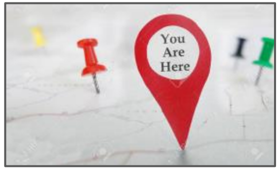
CSci 3081W: Program Design and Development

Lecture 04 - References and Pointers

Roadmap for Today

“The Stack” - Layer 1

“The Stack” - Layer N



References

`* & ->`

Pointers

The Call Stack (a.k.a. “The Stack”) keeps track of variables in memory.

Stack

First In

Last Out

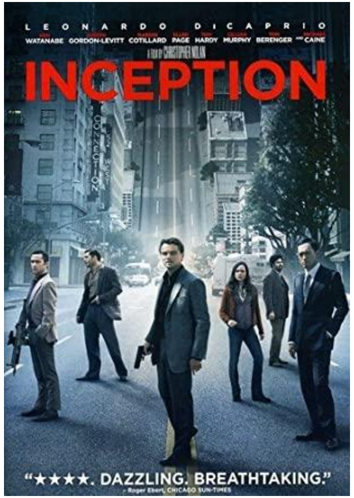
Queue

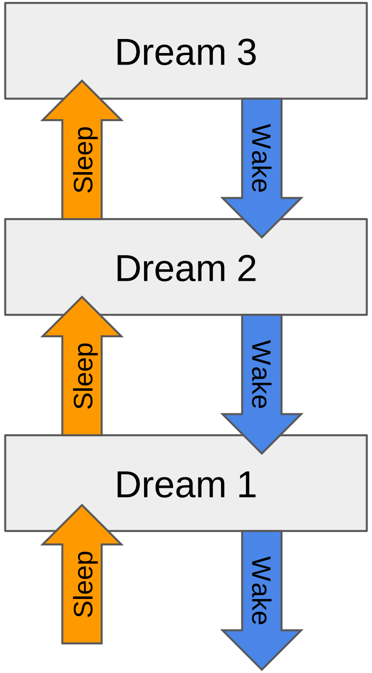
First In First Out

The Call Stack (a.k.a. “The Stack”) keeps track of variables in memory.

Stack

A Dream within a Dream


LEONARDO DICAPRIO
WATANABE GORDON-LEVITT COTILLARD PAGE HARDY MURPHY BERENGER CAINE
INCEPTION
“★★★★. DAZZLING. BREATHTAKING.”
- Roger Ebert, CHICAGO SUN-TIMES



Dream 3

Sleep

Wake

Dream 2

Sleep

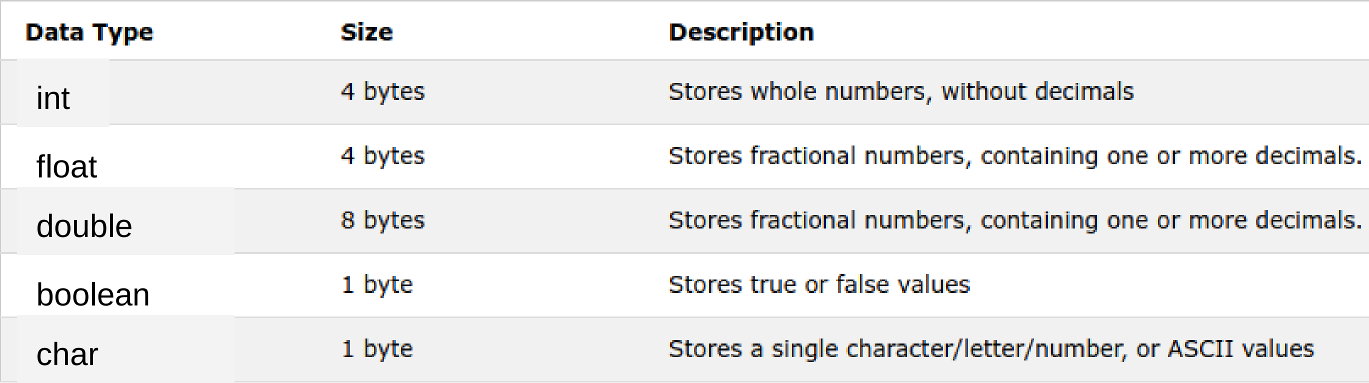
Wake

Dream 1

Sleep

Wake

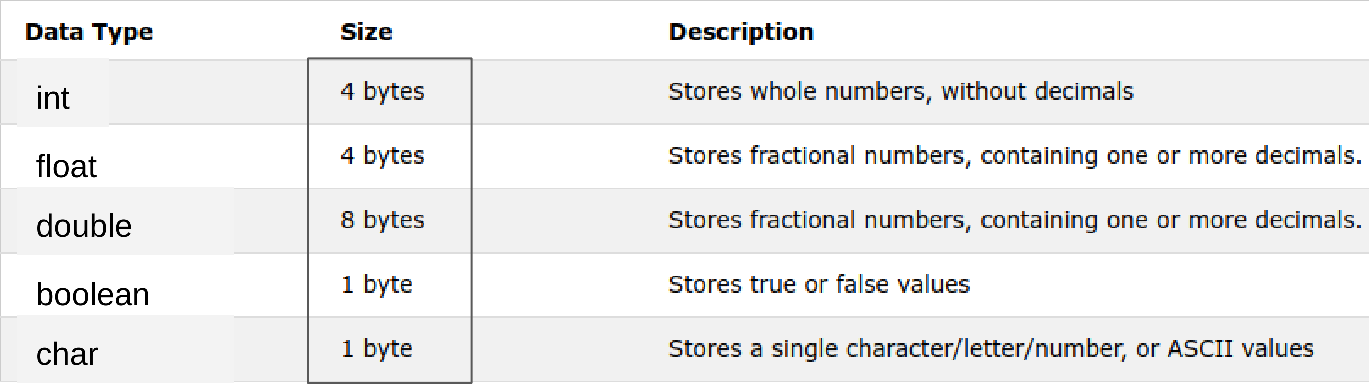
Basic Data Types



Data Type	Size	Description	Example
int	4 bytes	Stores whole numbers, without decimals	1, 2, 3, 4, -25, 0, 2343
float	4 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5001
double	8 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5000000001
boolean	1 byte	Stores true or false values	true, false, 0, 1
char	1 byte	Stores a single character/letter/number, or ASCII values	‘A’, ‘a’, ‘b’, ‘C’, ‘1’, ‘6’, ‘\n’

https://www.w3schools.com/cpp/cpp_data_types.asp

Basic Data Types

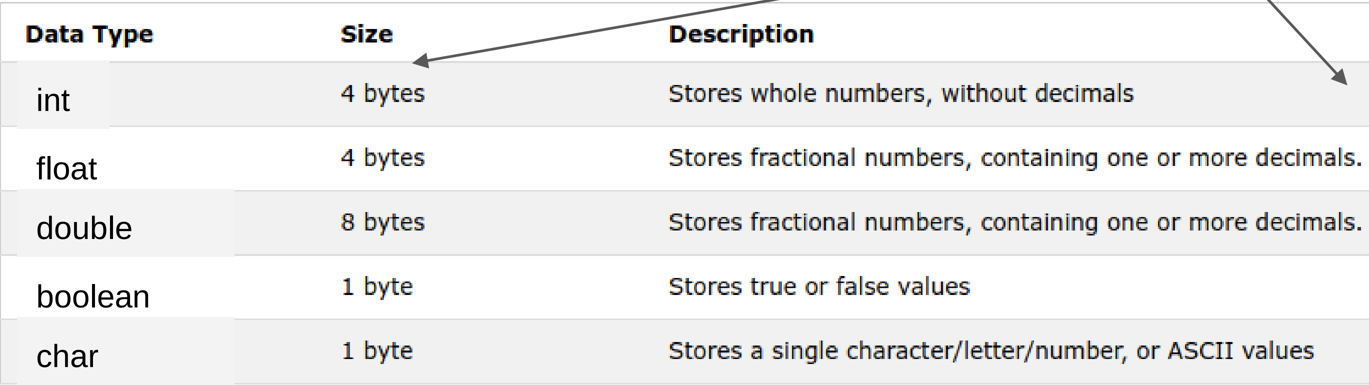


Data Type	Size	Description	Example
int	4 bytes	Stores whole numbers, without decimals	1, 2, 3, 4, -25, 0, 2343
float	4 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5001
double	8 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5000000001
boolean	1 byte	Stores true or false values	true, false, 0, 1
char	1 byte	Stores a single character/letter/number, or ASCII values	‘A’, ‘a’, ‘b’, ‘C’, ‘1’, ‘6’, ‘\n’

https://www.w3schools.com/cpp/cpp_data_types.asp

Basic Data Types

Maximum integer = ?

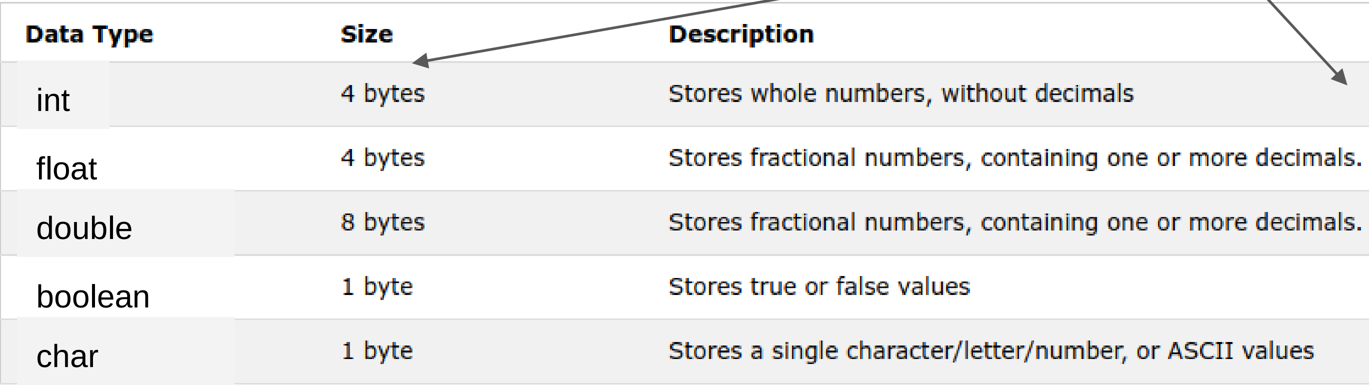


Data Type	Size	Description	Example
int	4 bytes	Stores whole numbers, without decimals	1, 2, 3, 4, -25, 0, 2343
float	4 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5001
double	8 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5000000001
boolean	1 byte	Stores true or false values	true, false, 0, 1
char	1 byte	Stores a single character/letter/number, or ASCII values	‘A’, ‘a’, ‘b’, ‘C’, ‘1’, ‘6’, ‘\n’

https://www.w3schools.com/cpp/cpp_data_types.asp

Basic Data Types

Maximum integer = 2^{32} **/ 2 - 1**



Data Type	Size	Description	Example
int	4 bytes	Stores whole numbers, without decimals	1, 2, 3, 4, -25, 0, 2343
float	4 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5001
double	8 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5000000001
boolean	1 byte	Stores true or false values	true, false, 0, 1
char	1 byte	Stores a single character/letter/number, or ASCII values	‘A’, ‘a’, ‘b’, ‘C’, ‘1’, ‘6’, ‘\n’

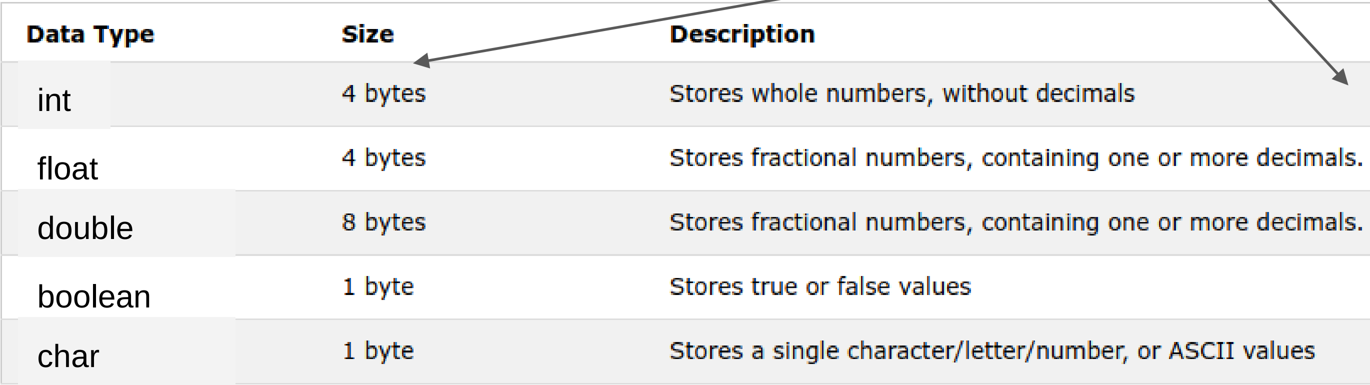
https://www.w3schools.com/cpp/cpp_data_types.asp

Basic Data Types

Maximum integer = 2^{32} **/ 2 - 1**= 4,294,967,296 **/ 2 - 1**

=

2,147,483,647



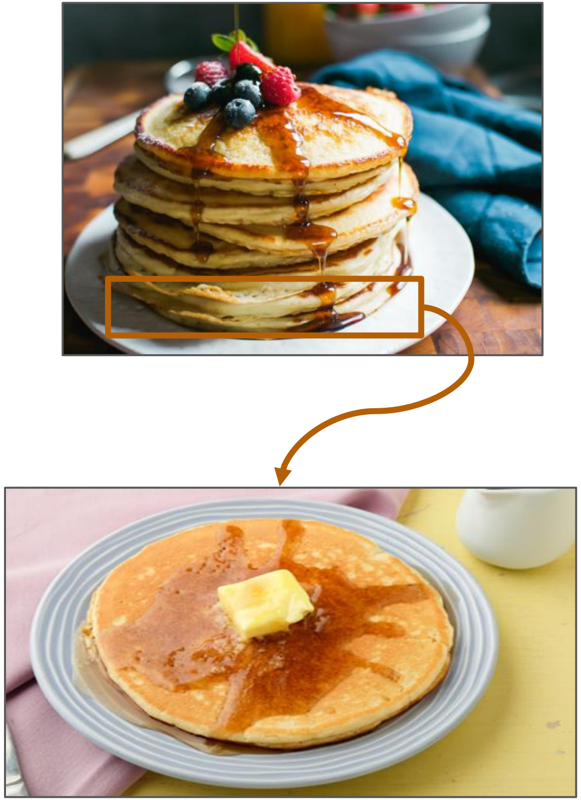
Data Type	Size	Description	Example
int	4 bytes	Stores whole numbers, without decimals	1, 2, 3, 4, -25, 0, 2343
float	4 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5001
double	8 bytes	Stores fractional numbers, containing one or more decimals.	3.14159, -1.5000000001
boolean	1 byte	Stores true or false values	true, false, 0, 1
char	1 byte	Stores a single character/letter/number, or ASCII values	‘A’, ‘a’, ‘b’, ‘C’, ‘1’, ‘6’, ‘\n’

https://www.w3schools.com/cpp/cpp_data_types.asp

Let’s take a look at the first layer of the stack.

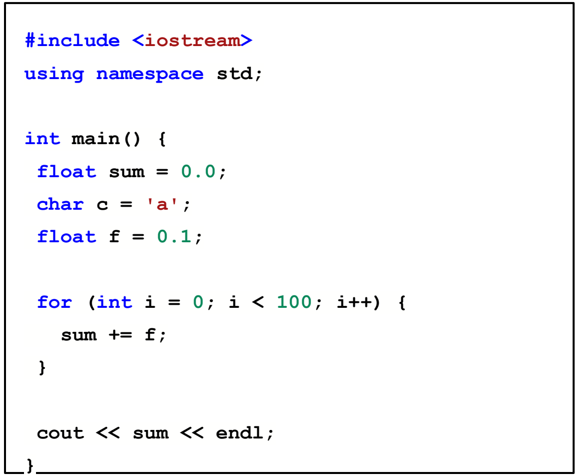
Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

Let’s take a look at the first layer of the stack.



Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

Let’s take a look at the first layer of the stack.



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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 100; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

Let’s take a look at the first layer of the stack.

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 100; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

float sum; (4 bytes)

char c; (1 byte)

float f; (4 bytes)

int i; (4 bytes)

Let’s take a look at the first layer of the stack.

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 100; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

Type	Variable	Content
float	sum	0.0
char	c	‘a’
float	f	0.1
int	i	0

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	0.0

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	**Variable**	**Content**
float	sum	0.0
char	**c**	**‘a’**

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	0.0
char	c	‘a’
float	**f**	**0.1**

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	**Variable**	**Content**
float	sum	0.0
char	c	‘a’
float	f	0.1
int	**i**	**0**

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	~~0.0~~ **0.1**
char	c	‘a’
float	f	0.1
int	i	0

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	0.1
char	c	‘a’
float	f	0.1
int	i	~~0~~ **1**

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	0.1
char	c	‘a’
float	f	0.1
int	i	1

1 < 2?

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	~~0.1~~ **0.2**
char	c	‘a’
float	f	0.1
int	i	1

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	**Variable**	**Content**
float	sum	0.2
char	c	‘a’
float	f	0.1
int	i	~~1~~ **2**

Programs use the stack to store variables and their values

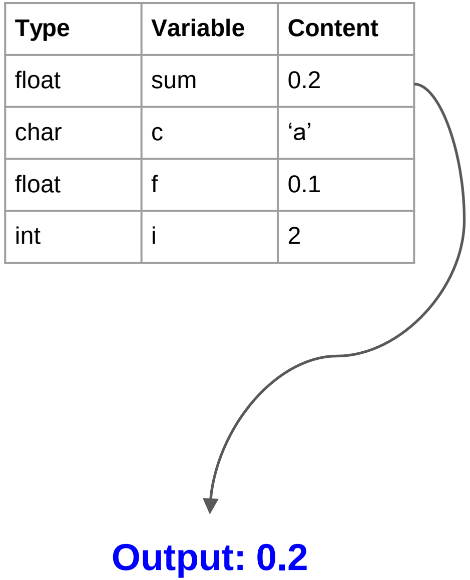
```
#include <iostream>
using namespace std;

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```

Type	Variable	Content
float	sum	0.2
char	c	‘a’
float	f	0.1
int	i	2

2 < 2?

Programs use the stack to store variables and their values

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

int main() {
 float sum = 0.0;
 char c = 'a';
 float f = 0.1;

 for (int i = 0; i < 2; i++) {
   sum += f;
 }

 cout << sum << endl;
}
```



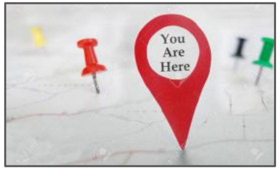
Type	Variable	Content
float	sum	0.2
char	c	‘a’
float	f	0.1
int	i	2

Output: 0.2

Roadmap for Today

“The Stack” - Layer 1

“The Stack” - Layer N

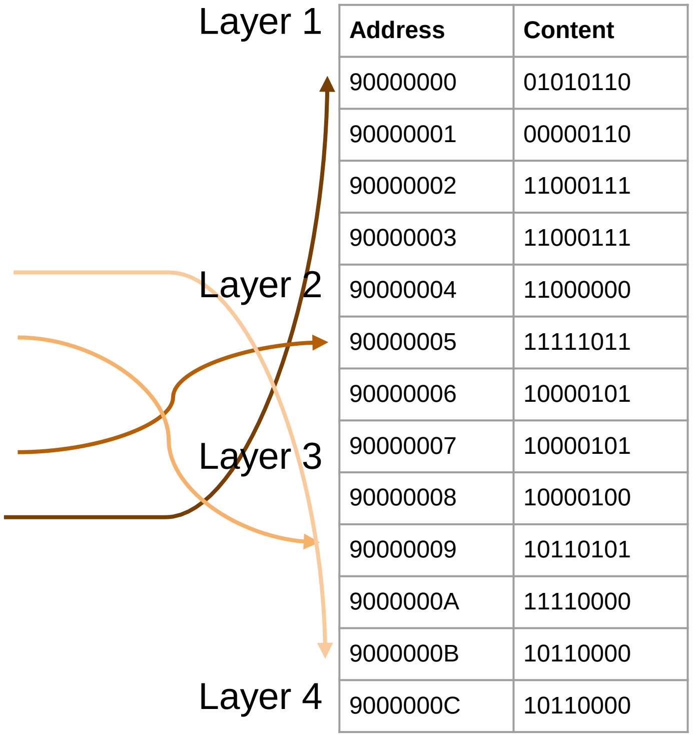


References

`* & ->`

Pointers

Each time a function is called the stack size increases



Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	11110000
9000000B	10110000
9000000C	10110000

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value			

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value	float	x	5.0

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value	float	x	5.0
3	add			

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value	float	x	5.0
3	add	float	a	5.0
3	add	float	b	5.0

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value	float	x	10.0
3	add	float	a	5.0
3	add	float	b	5.0
3	add	float	return	10.0

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Cout << 10 << endl;

Layer	Function	Type	Variable	Content
1	main	float	val	5.0
2	double_value	float	x	10.0
2	double_value	float	return	10.0

In order to understand the stack, we add two more columns.

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

float add(float a, float b);
float double_value(float x);

int main() {
  float val = 5.0;
  cout << double_value(val) << endl;
}

float add(float a, float b) {
  return a + b;
}

float double_value(float x) {
  x = add(x, x);
  return x;
}
```

Cout << 10 << endl;

Layer	Function	Type	Variable	Content
1	main	float	val	5.0

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value			

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value			

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value	float	x	2.0

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value	float	x	2.0
4	double_value			

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value	float	x	2.0
4	double_value	float	x	4.0

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value	float	x	2.0
4	double_value	float	x	4.0
5	double_value	float	x	8
6	double_value	float	x	16
7	double_value	float	x	32
8	double_value	float	x	64
9	double_value	float	x	128
10	double_value	float	x	256
11	double_value	float	x	512
...	...	...	...	...

What do you think will happen here?

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	Variable	Content
1	main	float	val	1.0
2	double_value	float	x	1.0
3	double_value	float	x	2.0
4	double_value	float	x	4.0
5	double_value	float	x	8
6	double_value	float	x	16
7	double_value	float	x	32
8	double_value	float	x	64
9	double_value	float	x	128
10	double_value	float	x	256
11	double_value	float	x	512
...	...	...	...	...

The call stack increases every time we call a function.

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

float double_value(float x);

int main() {
  float val = 1.0;
  val = double_value(val);
}

float double_value(float x) {
  return double_value(x+x);
}
```

Layer	Function	Type	ariable	Content
1	main			1.0
2	double_v			1.0
3				2.0
				4.0
			x	8
		oat	x	16
		float	x	32
	uble_value	float	x	64
	double_value	float	x	128
10	double_value	float	x	256
11	double_value	float	x	512
...	...	...	...	...

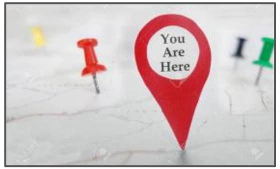
Stack Overflow!!!

(runs out of stack memory)

Roadmap for Today

“The Stack” - Layer 1

“The Stack” - Layer N



References

`* & ->`

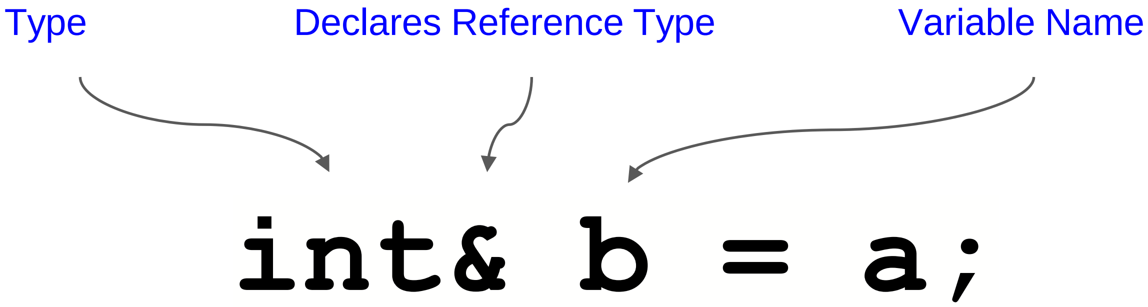
Pointers

References are like aliases for existing variables (e.g. int& b)

```
int a = 10;
```

References are like aliases for existing variables (e.g. int& b)

```
int a = 10;
```



Type

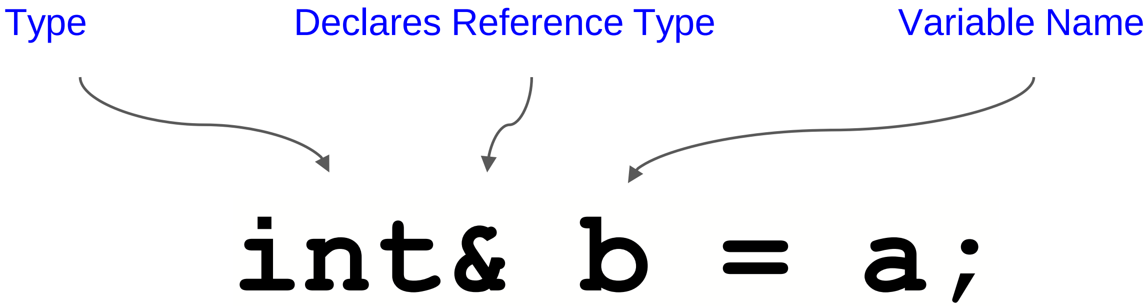
Declares Reference Type

Variable Name

```
int& b = a;
```

References are like aliases for existing variables (e.g. int**&** b)

```
int a = 10;
```



Type

Declares Reference Type

Variable Name

```
int& b = a;
```

References "point" to the same memory as another variable.

References are like aliases for existing variables (e.g. int& b)

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

int main()
{
            int a = 10;
            int& b = a;

            b++;
            cout << a << " " << b << endl;

            b = 32;
            // a = 32;
            cout << a << " " << b << endl;
            return 0;
}
```

b is another name for the variable **a**

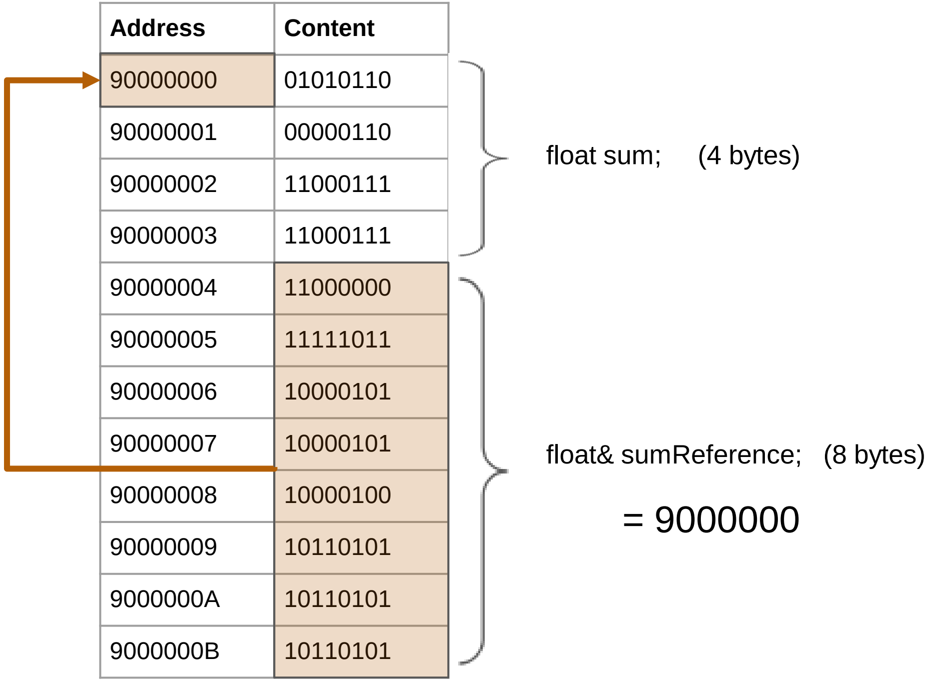
a and **b** *point* to the same memory!

When a change occurs in one, the other changes since it is the same memory being updated.

Why should we use them?

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

int main()
{
            int a = 10;
            int& b = a;

            b++;
            cout << a << " " << b << endl;

            a = 32;
            cout << a << " " << b << endl;
            return 0;
}
```

References are considered “safe” pointers with restrictions.



Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	10110101
9000000B	10110101

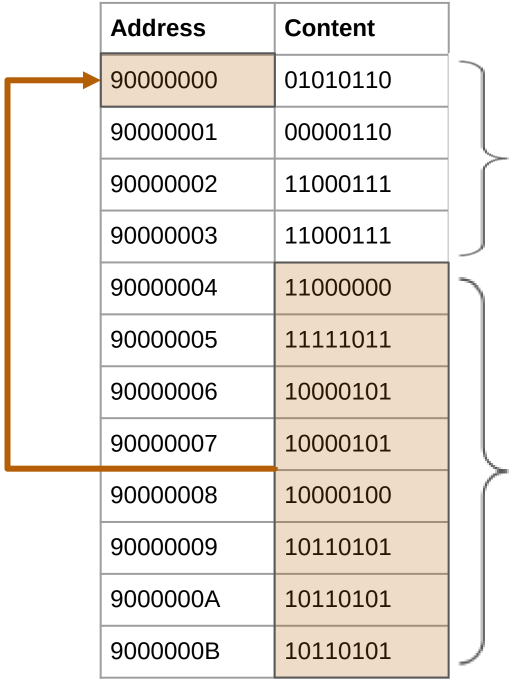
float sum;   (4 bytes)

float& sumReference;   (8 bytes)

= 9000000

A reference saves the address to a variable in memory.

References are considered “safe” pointers with restrictions.



Address	Content
90000000	01010110
90000001	00000110
90000002	11000111
90000003	11000111
90000004	11000000
90000005	11111011
90000006	10000101
90000007	10000101
90000008	10000100
90000009	10110101
9000000A	10110101
9000000B	10110101

float sum;   (4 bytes)

float& sumReference = sum;
(8 bytes)

= 9000000

A reference saves the address to a variable in memory.

<u>Restrictions</u>

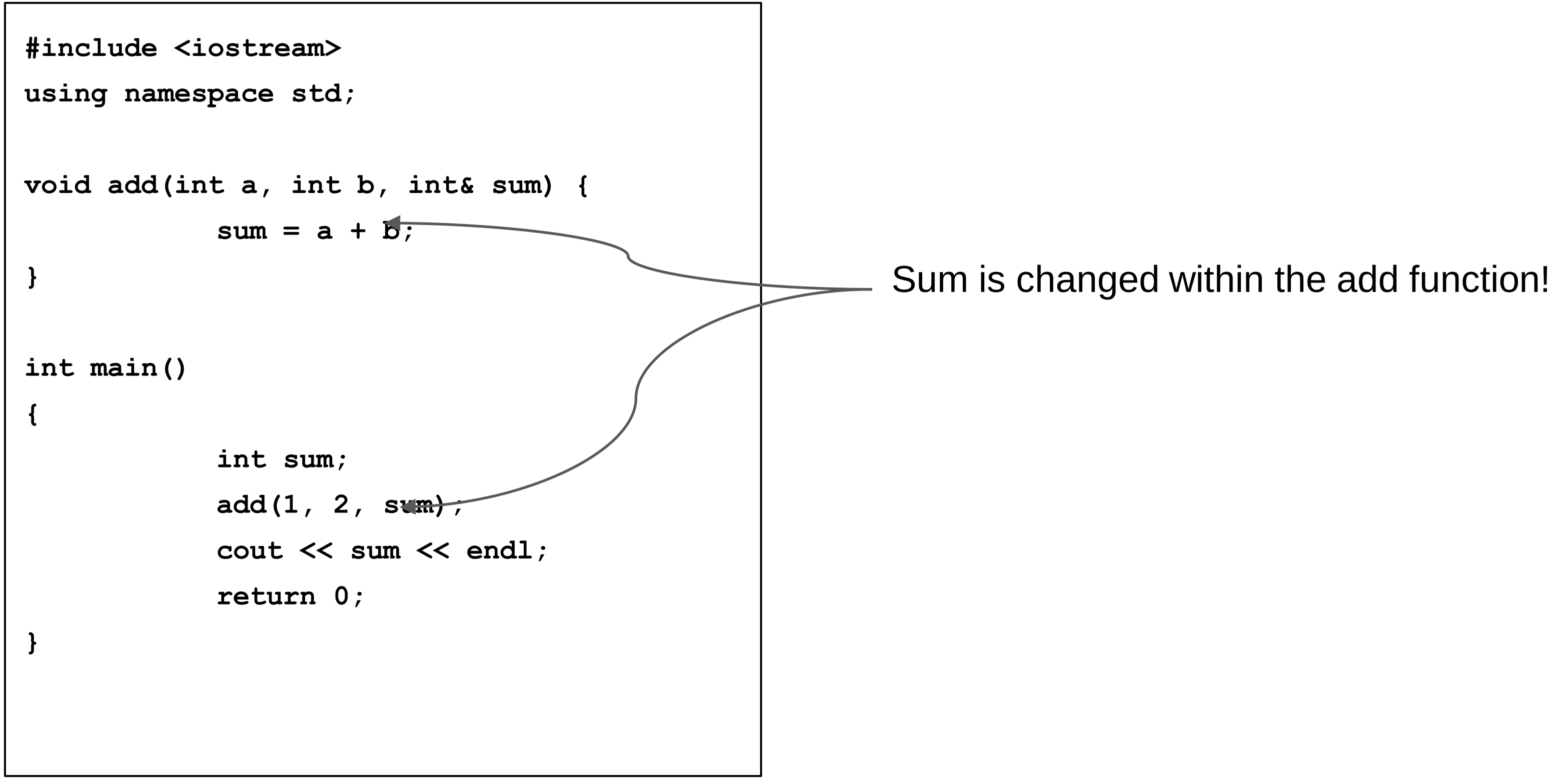
- The memory is assumed to exist!
  - Cannot point to invalid location
- It must be initialized from another variable.
- It cannot be changed.

Read Only Variables: The keyword **const** means that the variable cannot be changed.

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

int main()
{
        int a = 10;
        const int& b = a;

        b++;
        cout << a << " " << b << endl;

        a = 32;
        cout << a << " " << b << endl;
        return 0;
}
```

Fail!

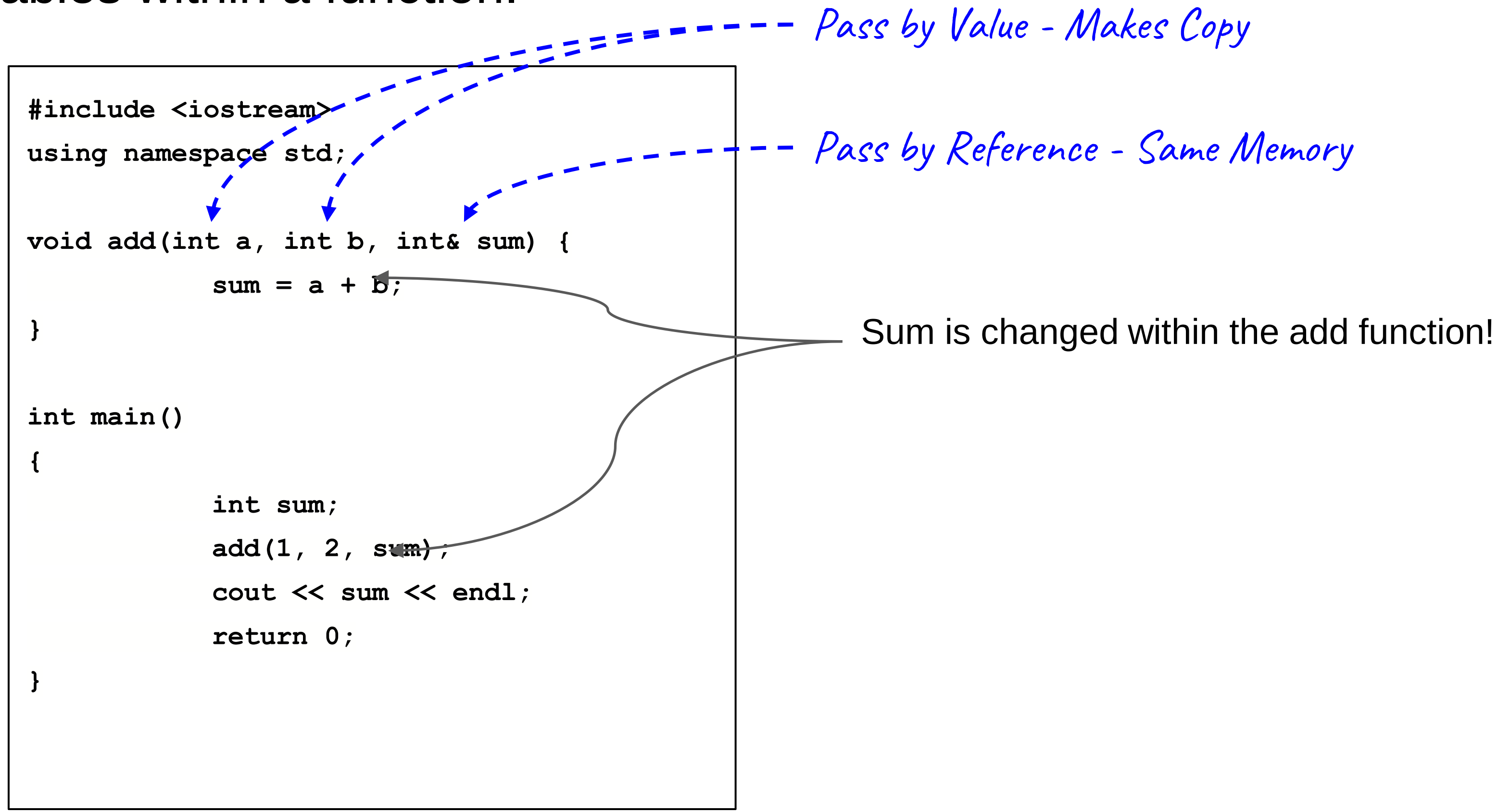
Output Parameters: *Passing by reference* allows you to change variables within a function.



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

void add(int a, int b, int& sum) {
            sum = a + b;
}

int main()
{
            int sum;
            add(1, 2, sum);
            cout << sum << endl;
            return 0;
}
```

Sum is changed within the add function!

Output Parameters: *Passing by reference* allows you to change variables within a function.



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

void add(int a, int b, int& sum) {
            sum = a + b;
}

int main()
{
            int sum;
            add(1, 2, sum);
            cout << sum << endl;
            return 0;
}
```

Pass by Value - Makes Copy

Pass by Reference - Same Memory

Sum is changed within the add function!

Key Takeaway: In functions, pass by reference if you would like to use parameters as output variables.

```
int divide(int a, int b, int& remainder) {
      remainder = a % b;
      return a / b;
}
```

Here we get the remainder from the parameter.

Multiple outputs!

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- Date of Birth
- Age
- Phone Number
- Email
- etc...

```
class Employee {
   ...
};


...


Employee jsmith;
```

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...

```
struct Employee {
  ...
};


...


Employee jsmith;
```

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...

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

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
};
```

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...
- **Manager**

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

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
  Employee manager; // ????
};
```

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...
- **Manager**

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

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
  Employee manager; ??
};
```

Incomplete Type!

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...
- **Manager**

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

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
  Employee& manager; // ????
};
```

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...
- **Manager**

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

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
  Employee& manager; /
};
```

Uninitialized Reference

Consider a data structure for an employee at a company using C++.

- Name
- Employee ID
- Salary
- **Date of Birth**
- Age
- **Phone Number**
- Email
- etc...
- **Manager**

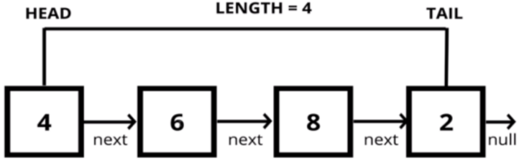
```
struct Date {
  int month;
  int day;
  int year;
};

struct PhoneNumber {
  int areaCode;
  int number;
};

struct Employee {
  char name[50];
  int id;
  float salary;
  Date dateOfBirth;
  int age;
  PhoneNumber phone;
  char email[100];
  Employee* manager;
};
```

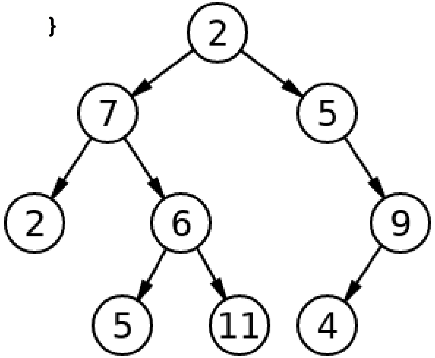
First example of where pointers are useful!

Common data structures like linked lists and binary trees use this technique.

```
// Linked List
struct Node {
            DataType
data;
            Node*
next;
}
```

```
// Binary Tree
struct Node {
            DataType
data;
            Node*
left;
            Node*
right;
}
```

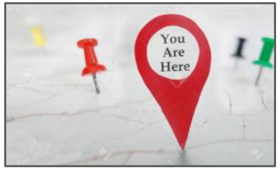




Roadmap for Today

“The Stack” - Layer 1

“The Stack” - Layer N



References

`* & ->`

Pointers

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```

Why is the output for a 11?

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```

Memory

Variable	Value

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```

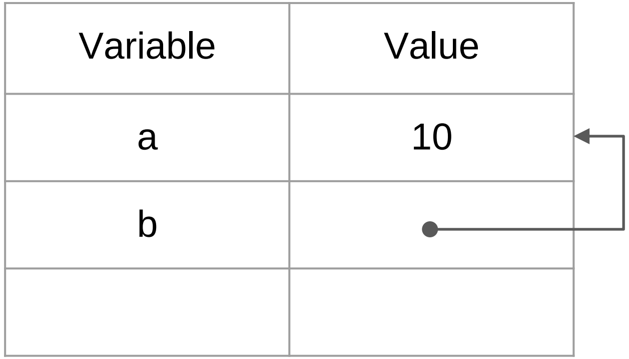
Memory

Variable	Value
a	10

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```

Memory

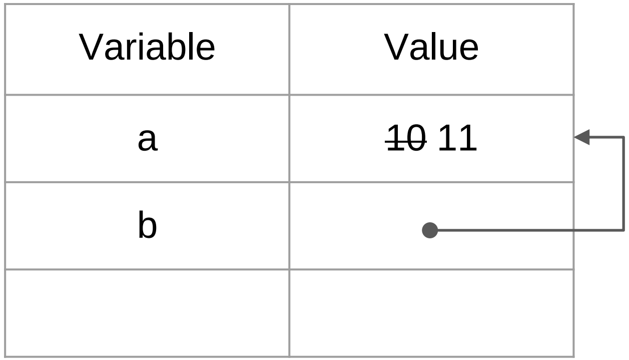


Variable	Value
a	10
b	●

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```

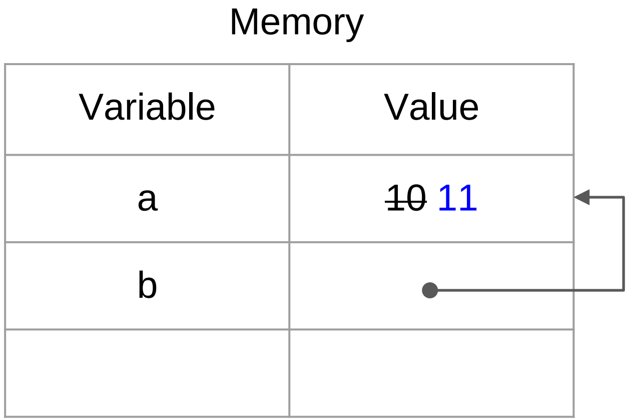
Memory



Variable	Value
a	~~10~~ 11
b	●

Recall: References use the same address as a variable.

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

int main()
{
            int a = 10;
            int& b = a;
            b++;

            cout << a << endl;
            return 0;
}
```



Memory

Variable	Value
a	~~10~~ 11
b	●

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
             return x+y;
}


int main()
{
             int a = 1;
             int b = 2;
             int sum;

             sum = add(a, b);

             cout << sum << endl;
             return 0;
}
```

Memory

Variable	Value

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
            return x+y;
}

int main()
{
            int a = 1;
            int b = 2;
            int sum;

            sum = add(a, b);

            cout << sum << endl;
            return 0;
}
```

Memory

Variable	Value
a	1

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
            return x+y;
}

int main()
{
            int a = 1;
            int b = 2;
            int sum;

            sum = add(a, b);

            cout << sum << endl;
            return 0;
}
```

Memory

Variable	Value
a	1
b	2

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
            return x+y;
}

int main()
{
            int a = 1;
            int b = 2;
            int sum;

            sum = add(a, b);

            cout << sum << endl;
            return 0;
}
```

Memory

Variable	Value
a	1
b	2
sum	?

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
            return x+y;
}

int main()
{
            int a = 1;
            int b = 2;
            int sum;

            sum = add(a, b);

            cout << sum << endl;
            return 0;
}
```

Memory

Variable	Value
a	1
b	2
sum	?
x	1
y	2

Notice the copy of a and b here

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
        return x+y;
}

int main()
{
        int a = 1;
        int b = 2;
        int sum;

        sum = add(a, b);

        cout << sum << endl;
        return 0;
}
```

Memory

Variable	Value
a	1
b	2
sum	~~?~~ 3
x	1
y	2

Recall: Passing by value makes a copy of the memory.

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

float add(int x, int y) {
            return x+y;
}

int main()
{
            int a = 1;
            int b = 2;
            int sum;

            sum = add(a, b);

            cout << sum << endl;
            return 0;
}
```

Memory

Variable	Value
a	1
b	2
sum	~~?~~ 3
x	1
y	2

Notice x and y no longer exist

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

Memory

Variable	Value

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

Memory

Variable	Value
a	1

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

Memory

Variable	Value
a	1
b	2

Recall: Passing by reference uses the same memory address.

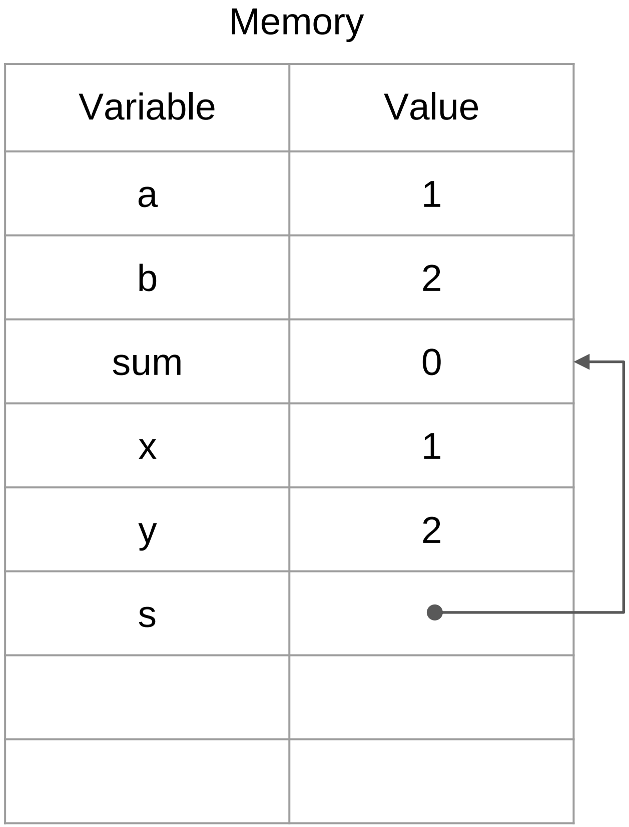
```
#include <iostream>
using namespace std;

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

Memory

Variable	Value
a	1
b	2
sum	0

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```



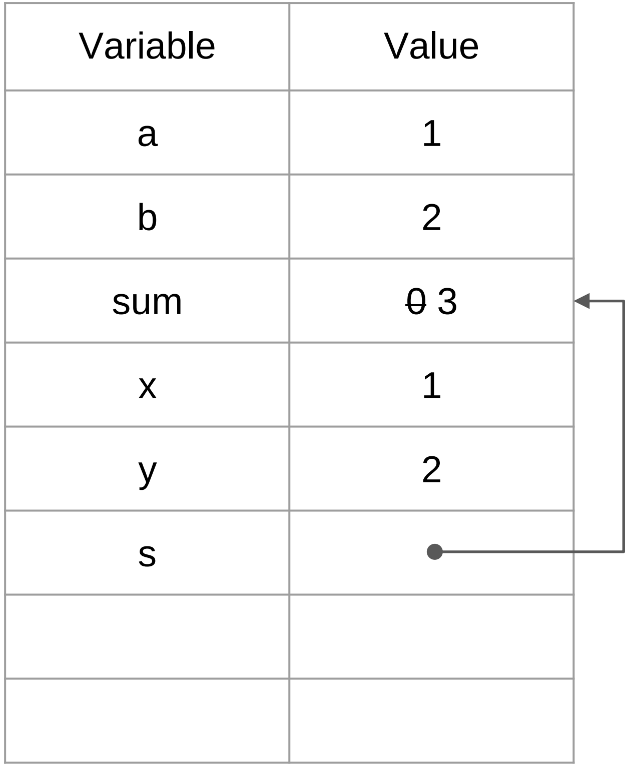
Memory

Variable	Value
a	1
b	2
sum	0
x	1
y	2
s	●

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

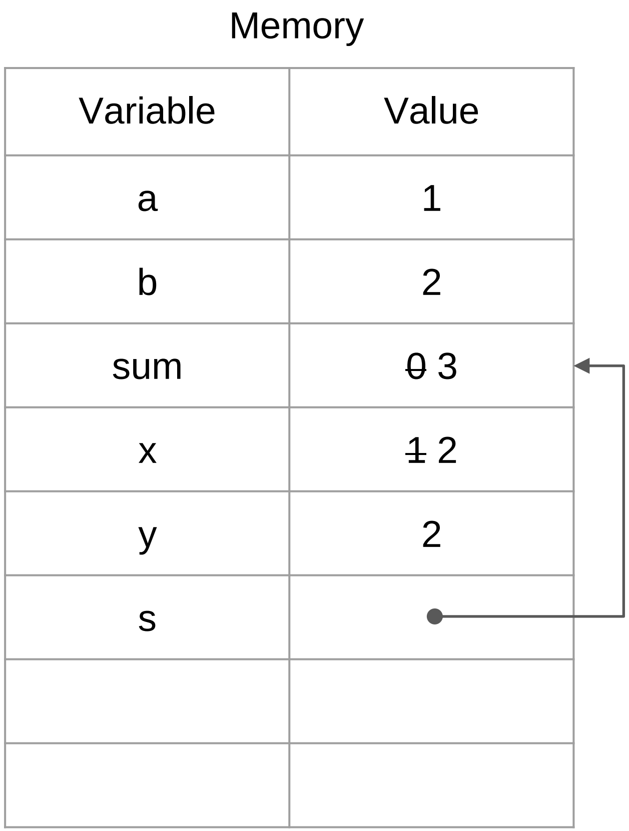
Memory



Variable	Value
a	1
b	2
sum	~~0~~ 3
x	1
y	2
s	●

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

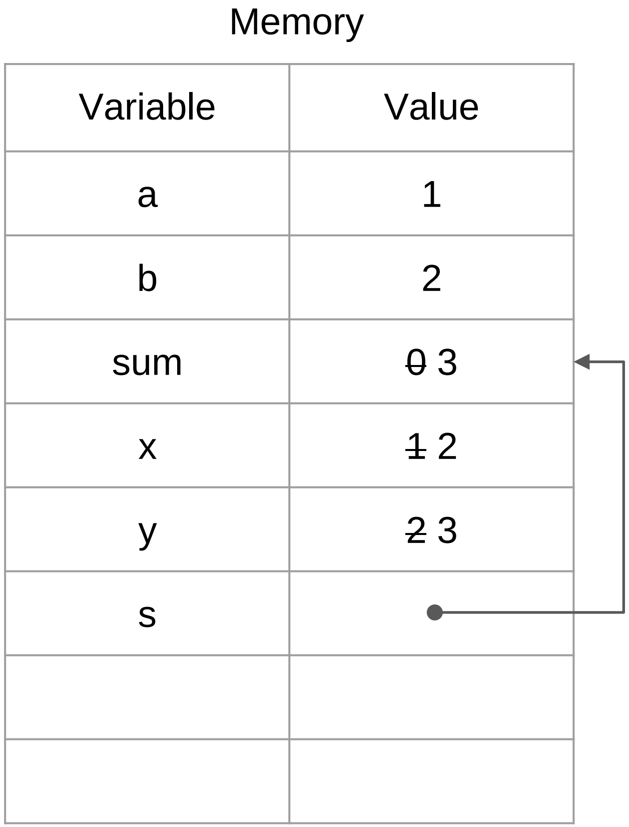


Memory

Variable	Value
a	1
b	2
sum	~~0~~ 3
x	~~1~~ 2
y	2
s	●

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```

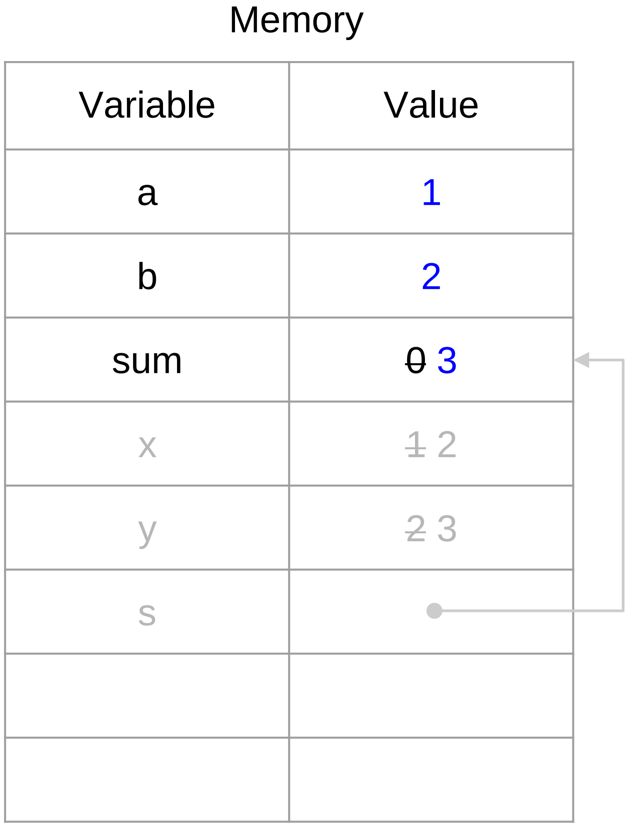


Memory

Variable	Value
a	1
b	2
sum	~~0~~ 3
x	~~1~~ 2
y	~~2~~ 3
s	●

Recall: Passing by reference uses the same memory address.

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

void add(int x, int y, int& s) {
              s = x+y;
              x++;
              y = x + 1;
}

int main()
{
              int a = 1;
              int b = 2;
              int sum = 0;

              add(a, b, sum);

              cout << a << " " << b << " " << sum <<
endl;
              return 0;
}
```



Memory

Variable	Value
a	1
b	2
sum	~~0~~ 3
x	~~1~~ 2
y	~~2~~ 3
s	●

What is a pointer?

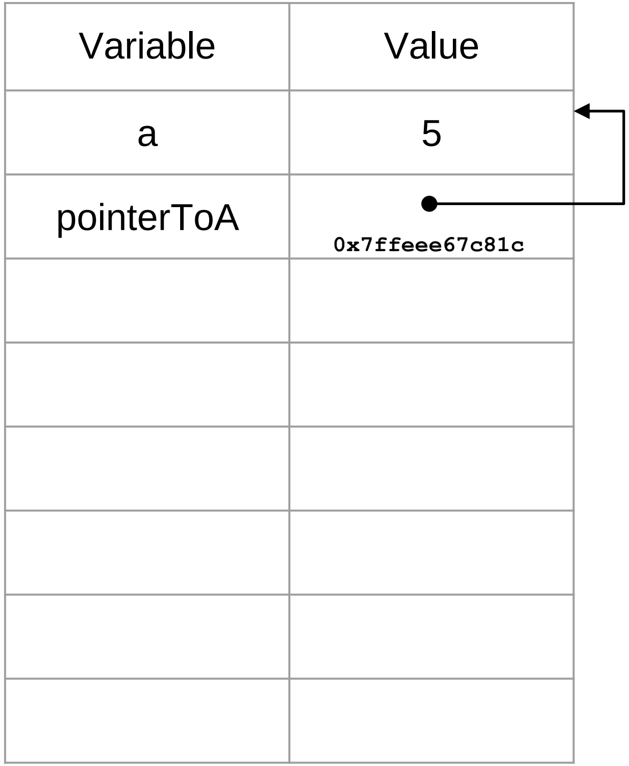
```
int main()
{
            int a = 5;
            int* pointerToA = &a;
}
```

A pointer is a memory address. That’s all folks...

```
int main()
{
            int a = 5;
            int* pointerToA = &a;

            cout << "a: " << a << endl;
            cout << "address of a: " << pointerToA <<
endl;
}
```

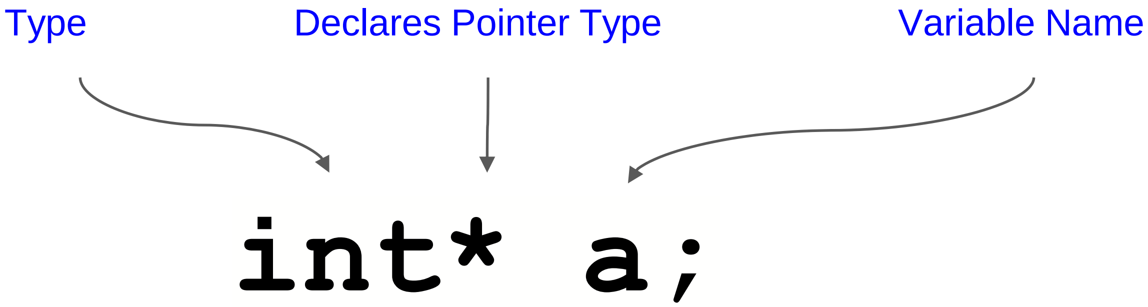
```
a: 5
address of a: 0x7ffeee67c81c
```

Memory



Variable	Value
a	5
pointerToA	● 0x7ffeee67c81c

A pointer is declared with the *



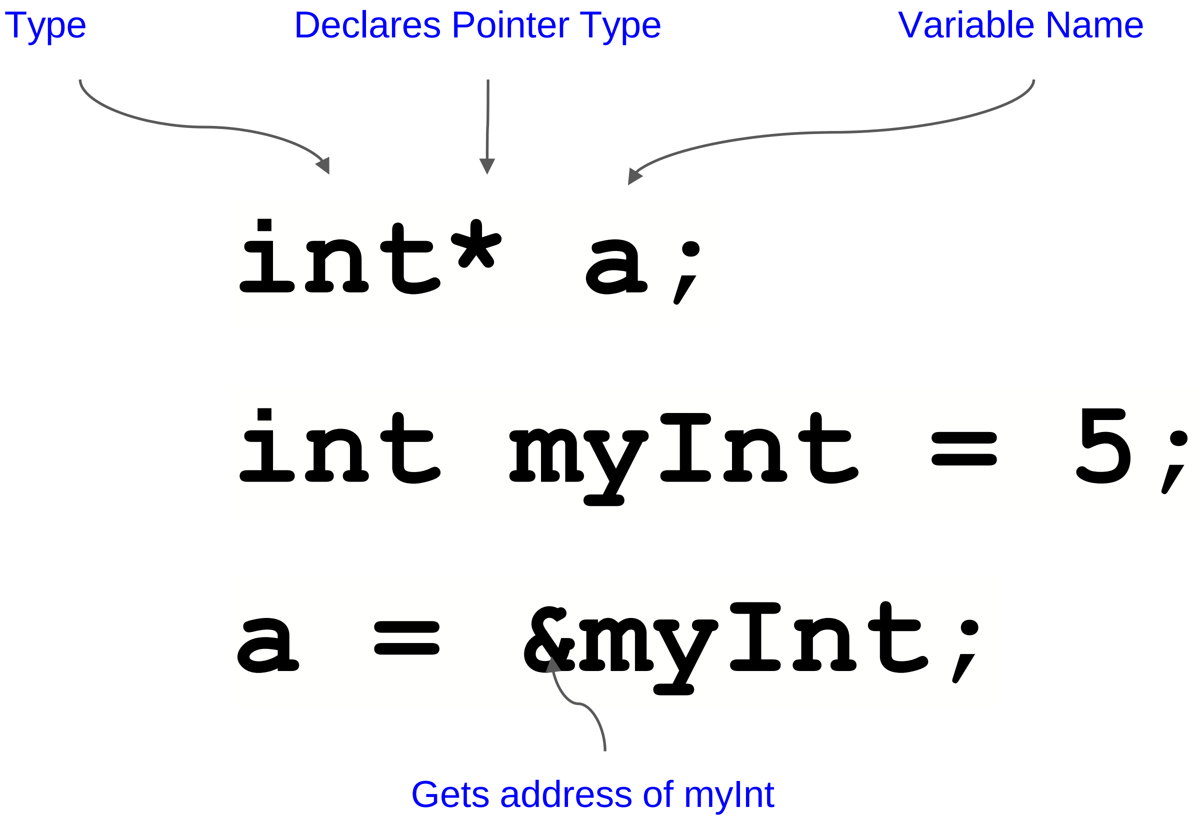
Type

Declares Pointer Type

Variable Name

```
int* a;
```

To get the address of an existing variable the & operator can be used.



Type

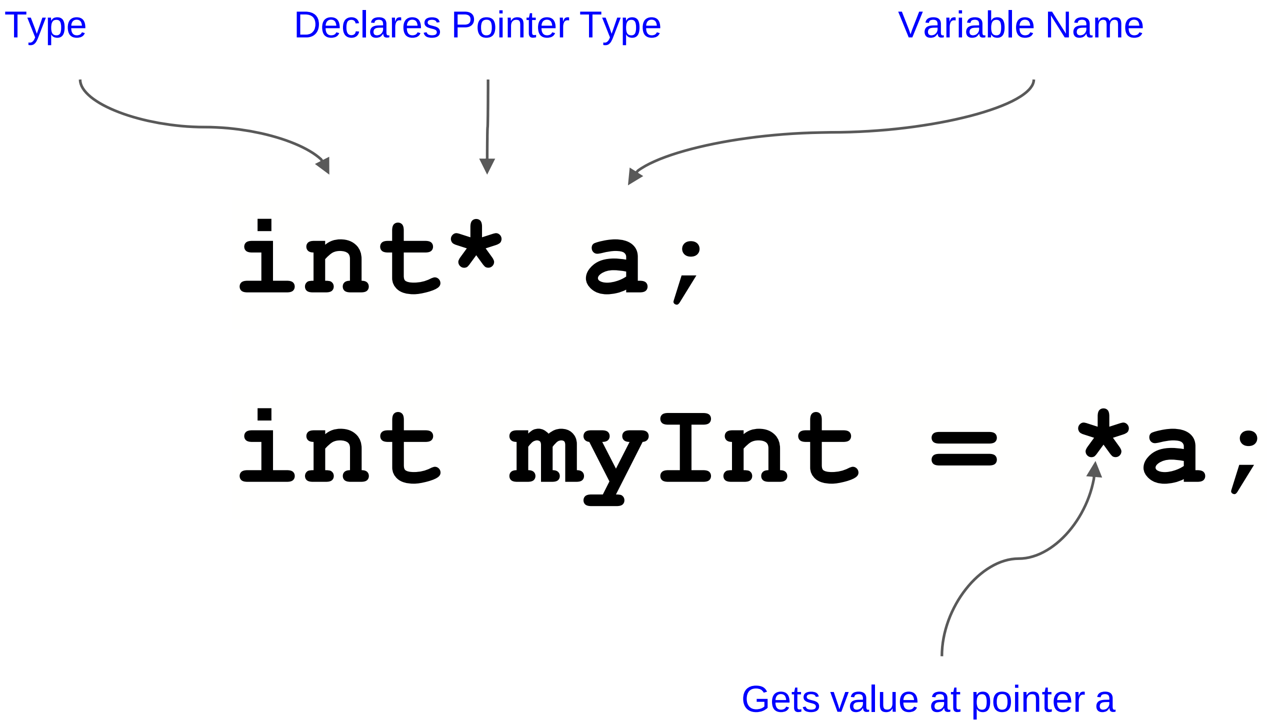
Declares Pointer Type

Variable Name

```
int* a;

int myInt = 5;

a = &myInt;
```

Gets address of myInt

To dereference a pointer to a value, you can use the * operator.



Type

Declares Pointer Type

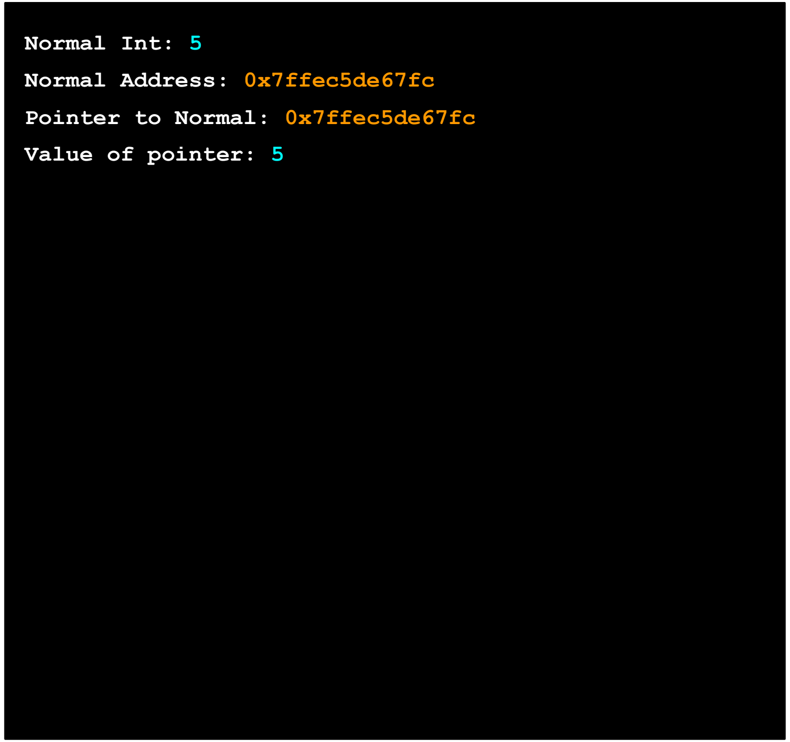
Variable Name

```
int* a;

int myInt = *a;
```

Gets value at pointer a

Below is an example of getting the memory address and printing the pointer value.

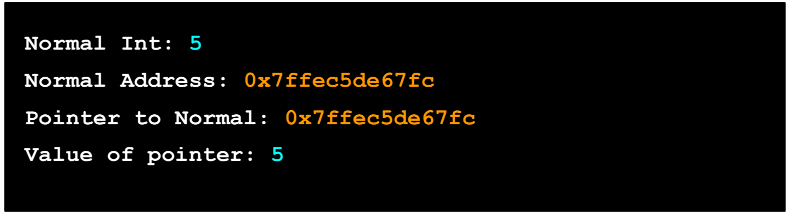
```
int main()
{
            // normal integer
            int normalInt = 5;

            // pointer to integer
            int* ptr = &normalInt;

            // print out values:
            cout << "Normal Int: " << normalInt <<
endl;
            cout << "Normal Address: " << &normalInt <<
endl;
            cout << "Pointer to Normal: " << ptr <<
endl;
            cout << "Value of pointer: " << *ptr <<
endl;
}
```



```
Normal Int: 5
Normal Address: 0x7ffec5de67fc
Pointer to Normal: 0x7ffec5de67fc
Value of pointer: 5
```

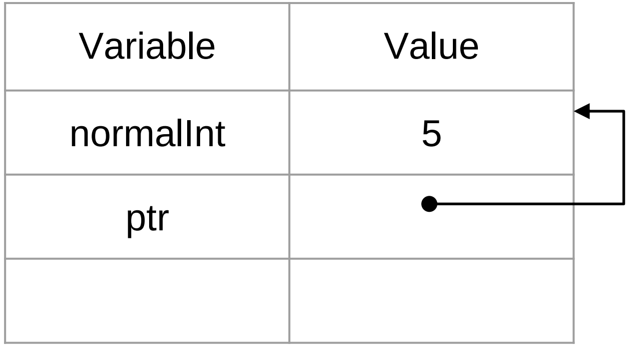
Below is an example of getting the memory address and printing the pointer value.

```
int main()
{
            // normal integer
            int normalInt = 5;

            // pointer to integer
            int* ptr = &normalInt;

            // print out values:
            cout << "Normal Int: " << normalInt <<
endl;
            cout << "Normal Address: " << &normalInt <<
endl;
            cout << "Pointer to Normal: " << ptr <<
endl;
            cout << "Value of pointer: " << *ptr <<
endl;
}
```



```
Normal Int: 5
Normal Address: 0x7ffec5de67fc
Pointer to Normal: 0x7ffec5de67fc
Value of pointer: 5
```

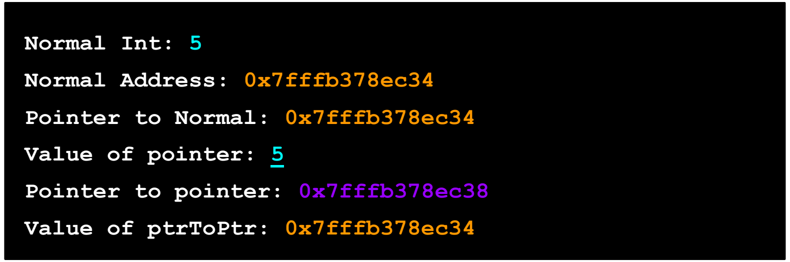
Memory



Variable	Value
normalInt	5
ptr	●

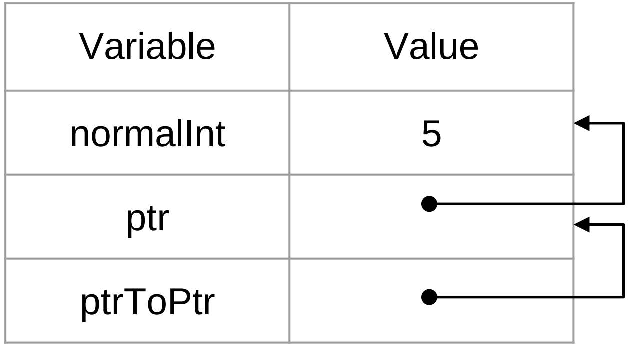
Pointers can point to other pointers and down the rabbit hole!

```
int main()
{
            // normal integer
            int normalInt = 5;

            // pointer to integer
            int* ptr = &normalInt;

            // pointer to pointer
            int** ptrToPtr = &ptr;

            // print out values:
            cout << "Normal Int: " << normalInt << endl;
            cout << "Normal Address: " << &normalInt <<
endl;
            cout << "Pointer to Normal: " << ptr << endl;
            cout << "Value of pointer: " << *ptr << endl;
            cout << "Pointer to pointer: " << ptrToPtr <<
endl;
            cout << "Value of ptrToPtr: " << *ptrToPtr <<
```



```
Normal Int: 5
Normal Address: 0x7fffb378ec34
Pointer to Normal: 0x7fffb378ec34
Value of pointer: 5
Pointer to pointer: 0x7fffb378ec38
Value of ptrToPtr: 0x7fffb378ec34
```

Memory



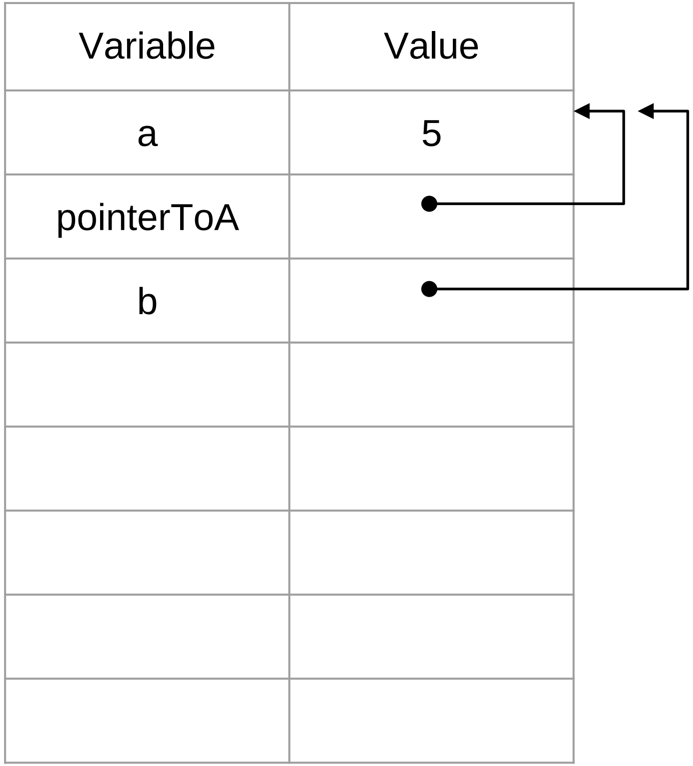
Variable	Value
normalInt	5
ptr	●
ptrToPtr	●

What is the key difference between a **pointer** and a **reference**?

```
int main()
{
             int a = 5;
             int* pointerToA = &a;
             int &b = a;

             cout << "a: " << a << endl;
             cout << "address of a: " << pointerToA <<
endl;
             cout << "b: " << b << endl;
             cout << "address of b: " << &b << endl;
}
```

```
a: 5
address of a: 0x7ffeee67c81c
b: 5
address of b: 0x7ffeee67c81c
```

Memory



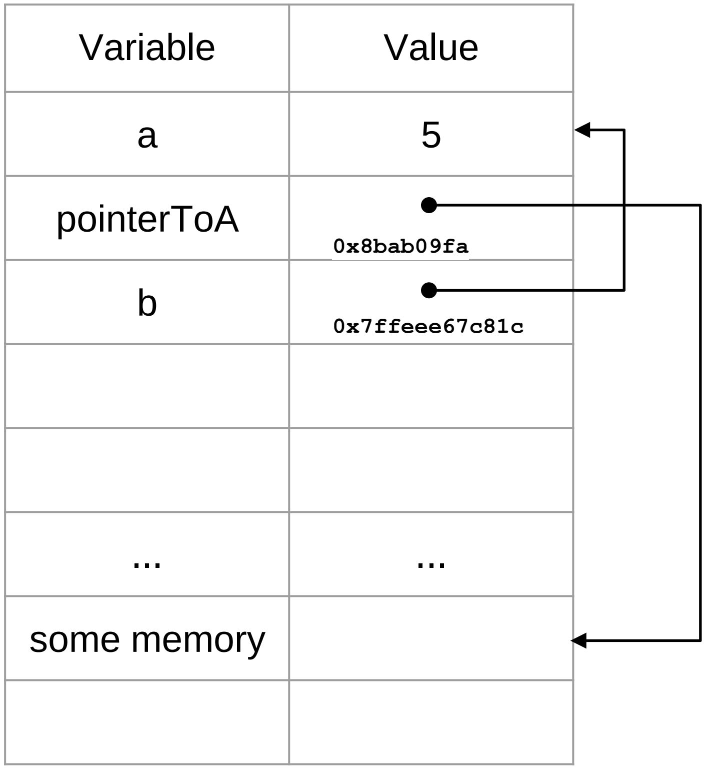
Variable	Value
a	5
pointerToA	●
b	●

A pointer is a memory address and a **reference** is a valid memory address.

```
int main()
{
              int a = 5;
              int* pointerToA = (int*)2343242234;
              int &b = a;

              cout << "a: " << a << endl;
              cout << "pointer to some memory: " << pointerToA <<
endl;
              cout << "b: " << b << endl;
              cout << "address of b: " << &b << endl;
}
```

```
a: 5
pointer to some memory:: 0x8bab09fa
b: 5
address of b: 0x7ffeee67c81c
```

Memory

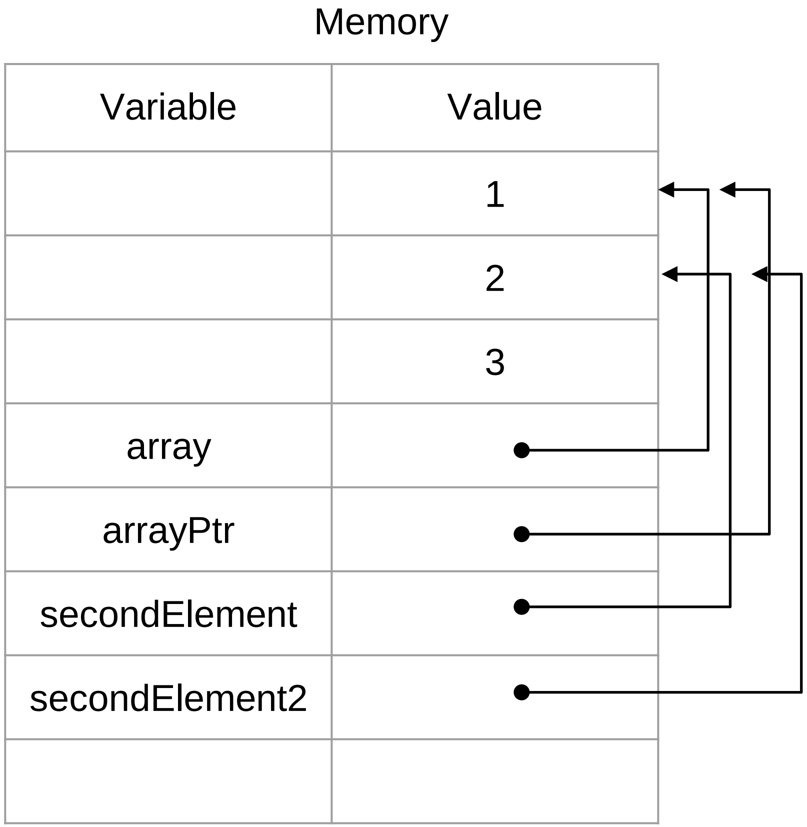


Variable	Value
a	5
pointerToA	0x8bab09fa
b	0x7ffeee67c81c
...	...
some memory	

Arrays can be converted to pointers since they are pointers.

```
int array[] = {1, 2, 3};
int* arrayPtr = array;
```

```
int main()
{
		int array[] = {1, 2, 3};
		int* arrayPtr = array;
		int* secondElement = &array[1];
		int* secondElement2 = arrayPtr + 1;

		cout << *secondElement << endl;
}
```



Memory

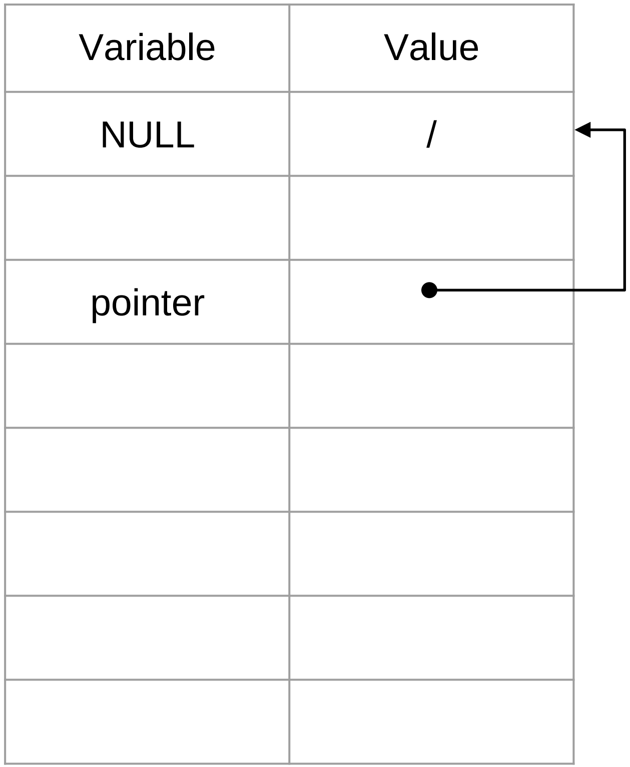
Variable	Value
	1
	2
	3
array	●
arrayPtr	●
secondElement	●
secondElement2	●

NULL or nullptr is a memory address that allows us to point to nothing.

```
int main()
{
            int* pointer = NULL; // can also use nullptr

            cout << "pointer: " << pointer << endl;
            cout << *pointer << endl;
}
```

```
pointer: 0x0000000
exited, segmentation fault
```

Memory



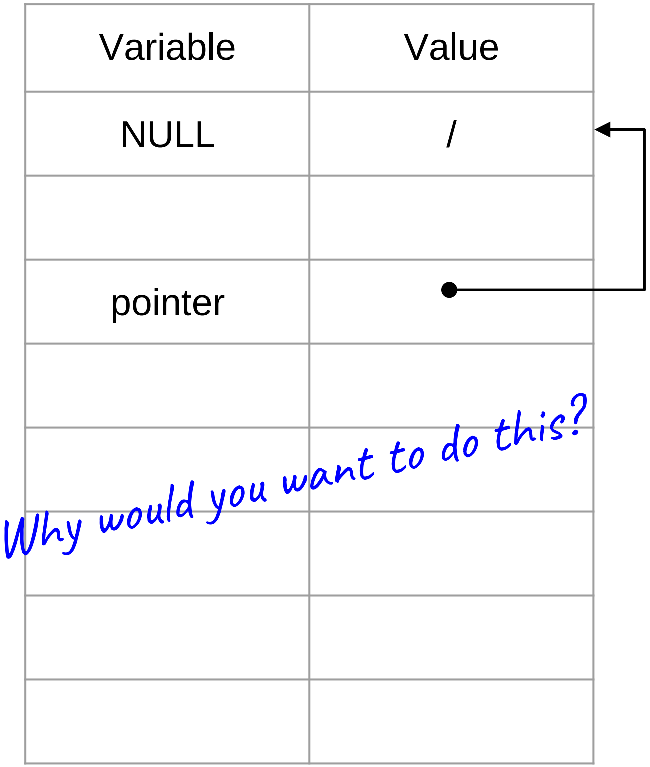
Variable	Value
NULL	/
pointer	●

NULL or nullptr is a memory address that allows us to point to nothing.

```
int main()
{
        int* pointer = NULL; // can also use nullptr

        cout << "pointer: " << pointer << endl;
        cout << *pointer << endl;
}
```

```
pointer: 0x0000000
exited, segmentation fault
```

Memory



Variable	Value
NULL	/
pointer	

Why would you want to do this?

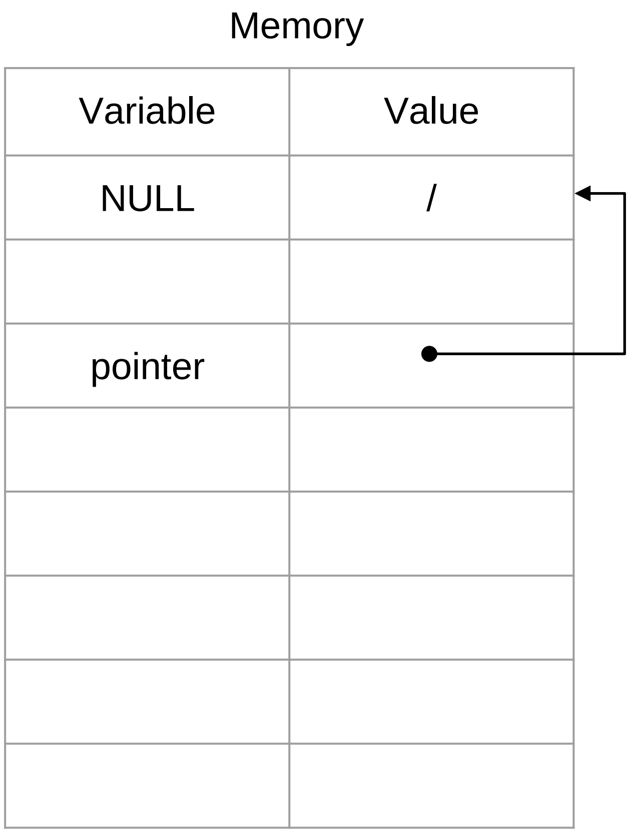
NULL or nullptr is a memory address that allows us to point to nothing.

```
int main()
{
              int* pointer = NULL; // can also use nullptr

              cout << "pointer: " << pointer << endl;
              cout << *pointer << endl;
}
```

This is what causes the dreaded segfault:

- Accessing invalid memory

```
pointer: 0x0000000
exited, segmentation fault
```



Memory

Variable	Value
NULL	/
pointer	●

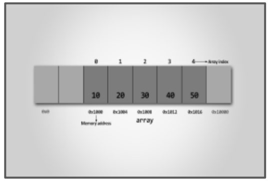
To access class variables and methods, use the -> operator.

```
class Vector3 {
public:
            float x, y, z;
}

int main() {
      Vector3 vec;
      Vector3* vecPtr = &vec;
      Vec.x = 0.0;
      vecPtr->x = 1.0;
            return 0;
}
```

The Arrow operator “->” allows us access class members and methods.

This is the same as: *(vecPtr).x

Roadmap for Today

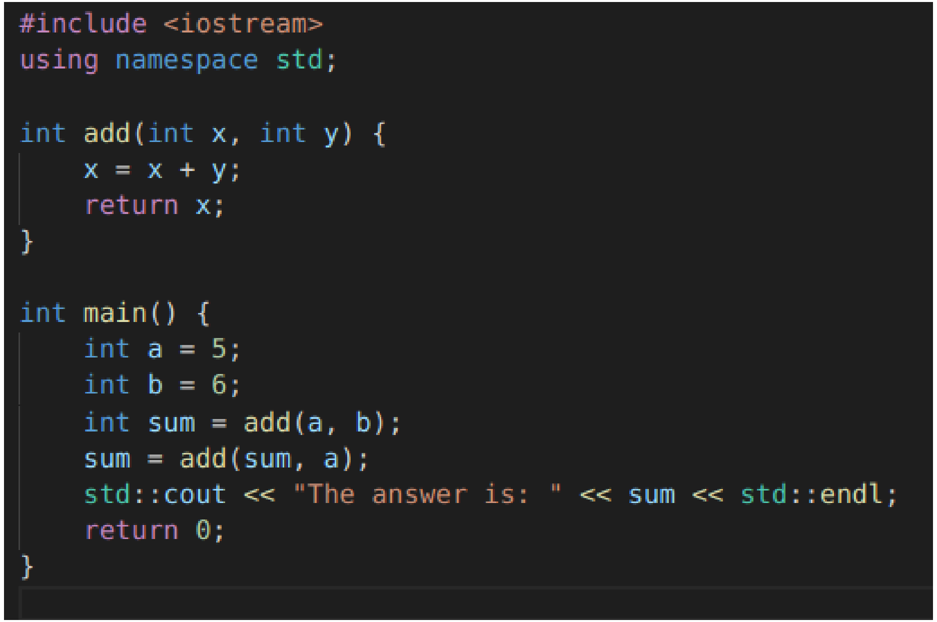


Practice: Diagramming Memory

Dynamic Memory

Classes and Dynamic Memory

Example 1: Pass by Value

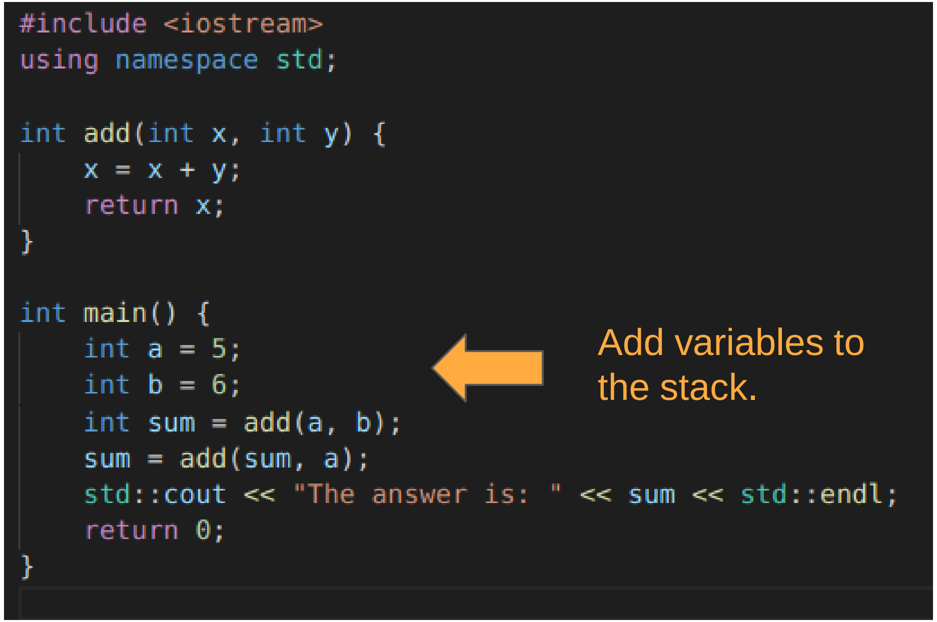


```cpp
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Memory

Type	Name	Value

Example 1: Pass by Value



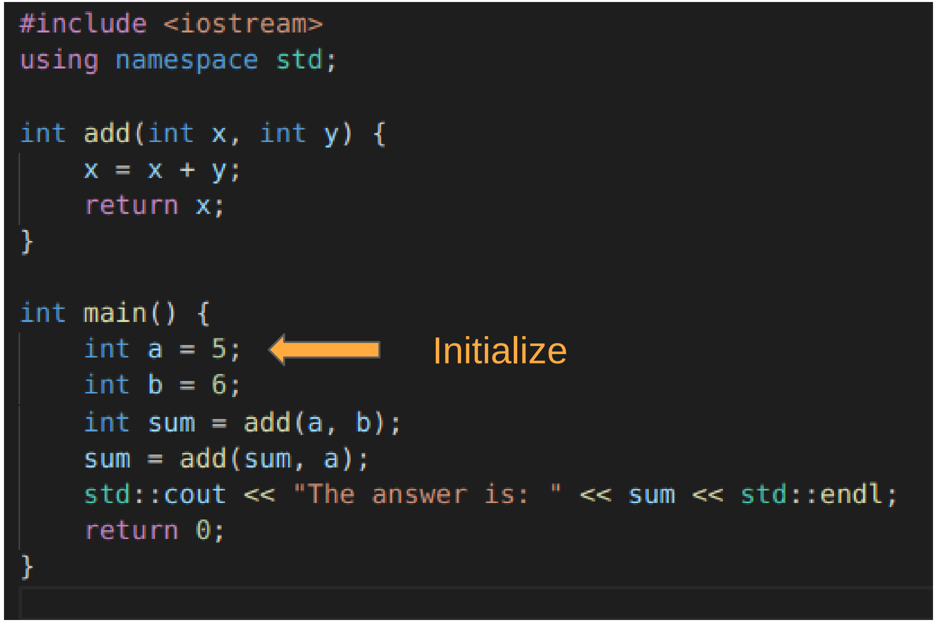
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Add variables to the stack.

Memory

Type	Name	Value
int	**a**	**?**
int	**b**	**?**
int	**sum**	**?**

Example 1: Pass by Value



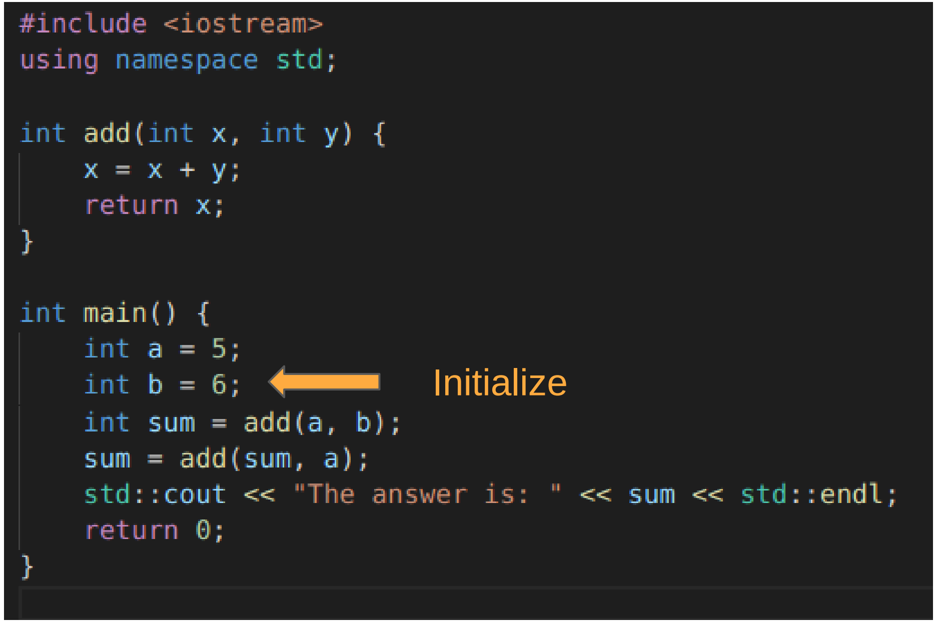
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Initialize

Memory

Type	Name	Value
int	a	~~?~~ **5**
int	b	?
int	sum	?

Example 1: Pass by Value



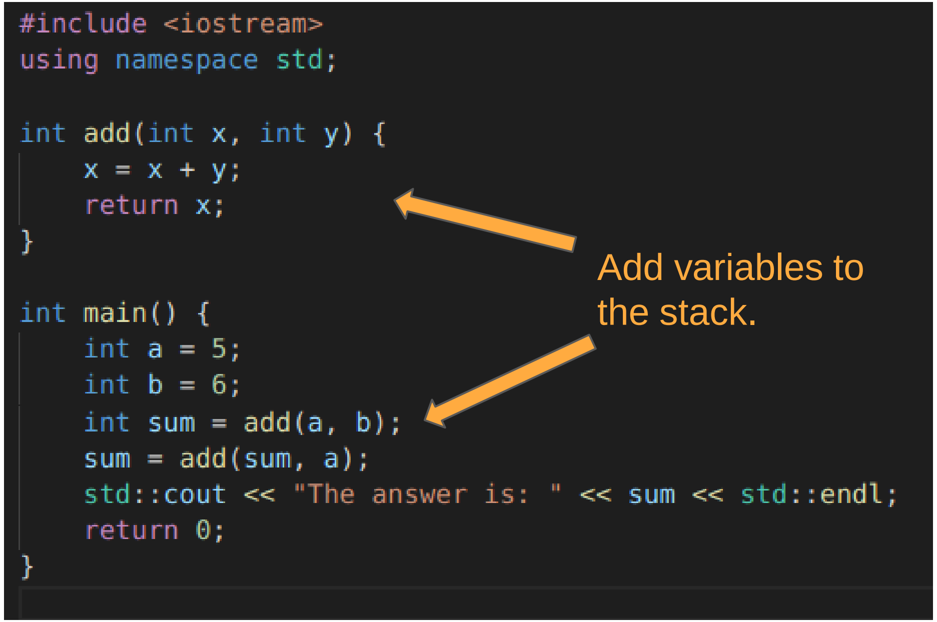
```cpp
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Initialize

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ **6**
int	sum	?

Example 1: Pass by Value



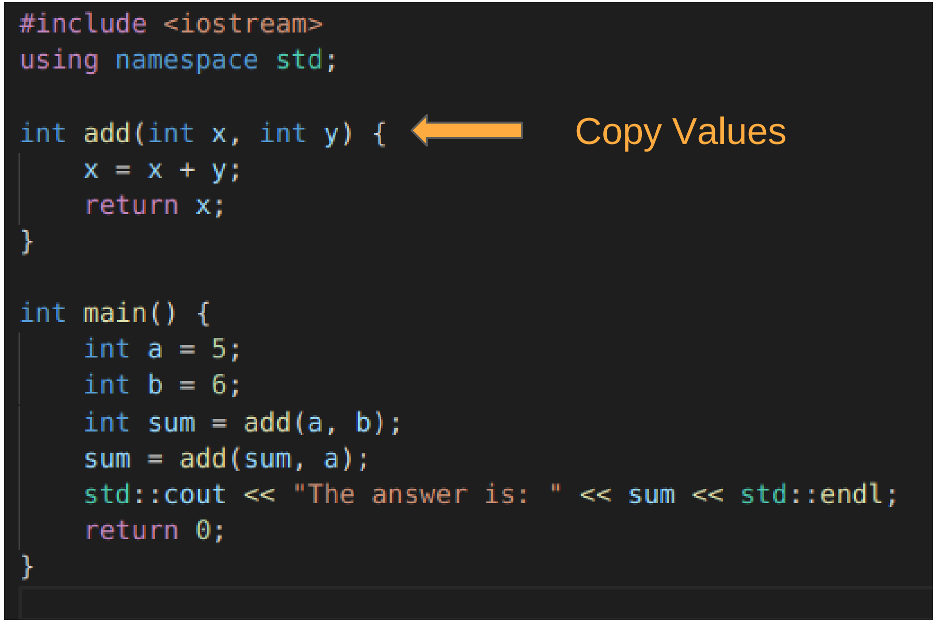
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Add variables to the stack.

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	?
int	**x**	**?**
int	**y**	**?**

Example 1: Pass by Value



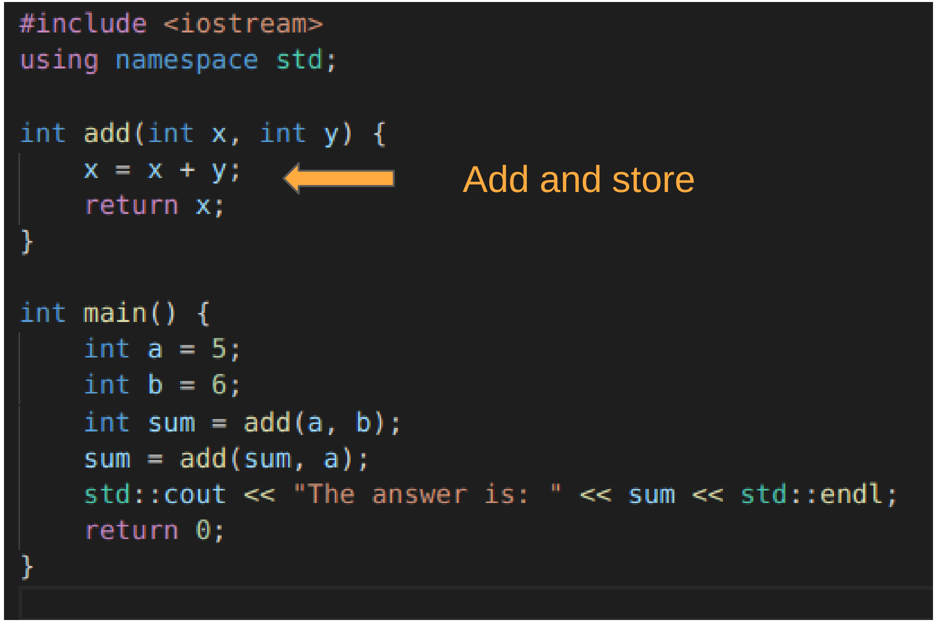
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Copy Values

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	?
int	x	**~~?~~ 5**
int	y	**~~?~~ 6**

Example 1: Pass by Value



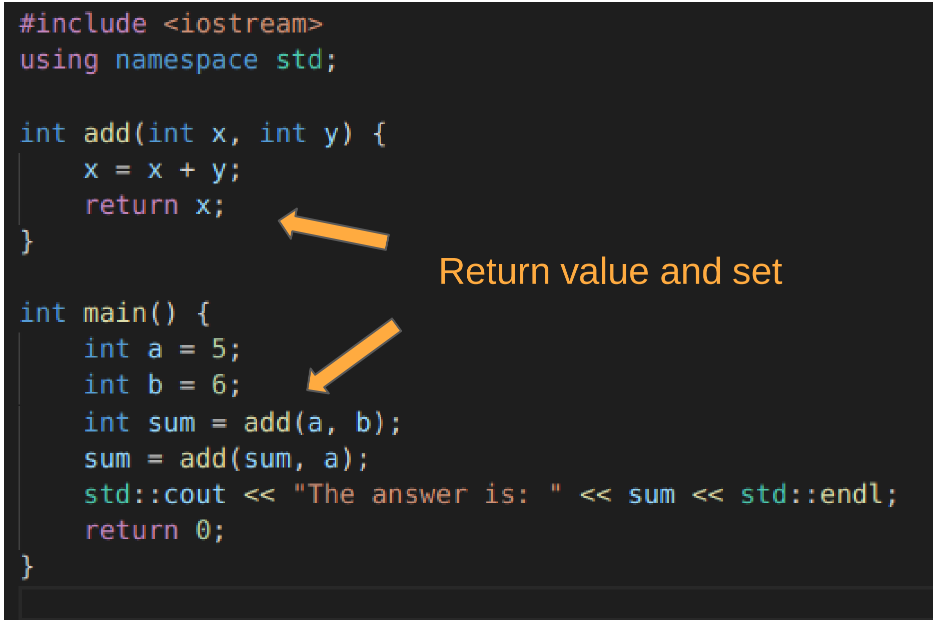
```cpp
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Add and store

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	?
int	x	~~?~~ ~~**5**~~ **11**
int	y	~~?~~ 6

Example 1: Pass by Value



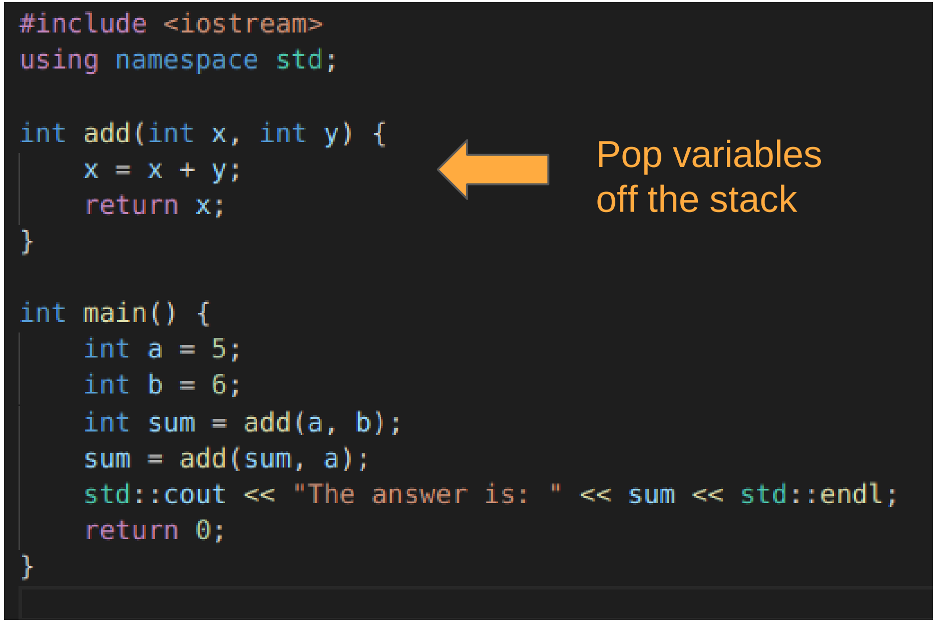
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Return value and set

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	**~~?~~ 11**
int	x	~~?~~ ~~5~~ 11
int	y	~~?~~ 6

Example 1: Pass by Value



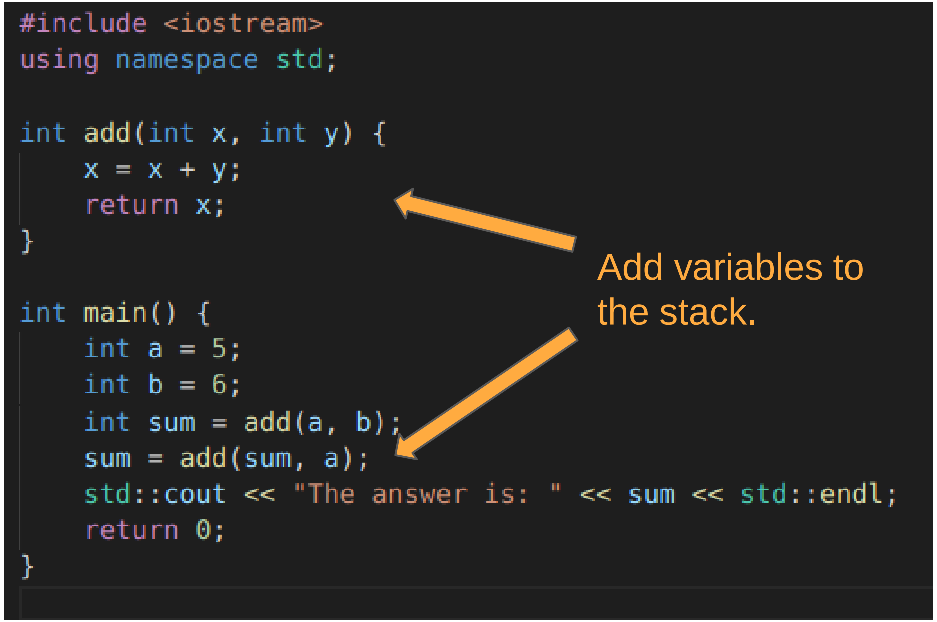
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Pop variables off the stack

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	~~?~~ 11
		~~? 5~~ 11
		~~?~~ 6

Example 1: Pass by Value



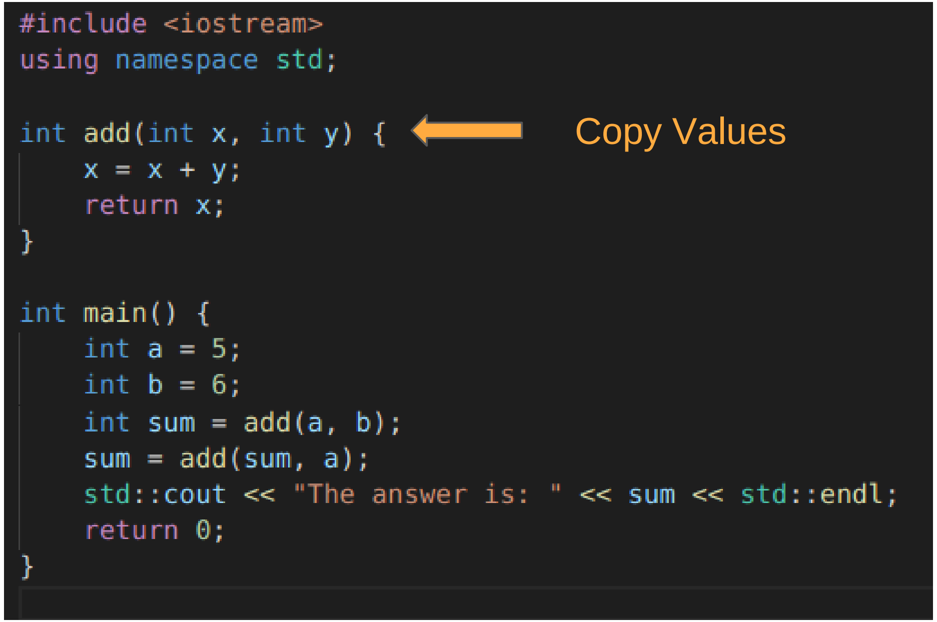
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Add variables to the stack.

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	~~?~~ 11
int	**x**	~~?~~ ~~5~~ 11
int	**y**	~~?~~ 6

Example 1: Pass by Value



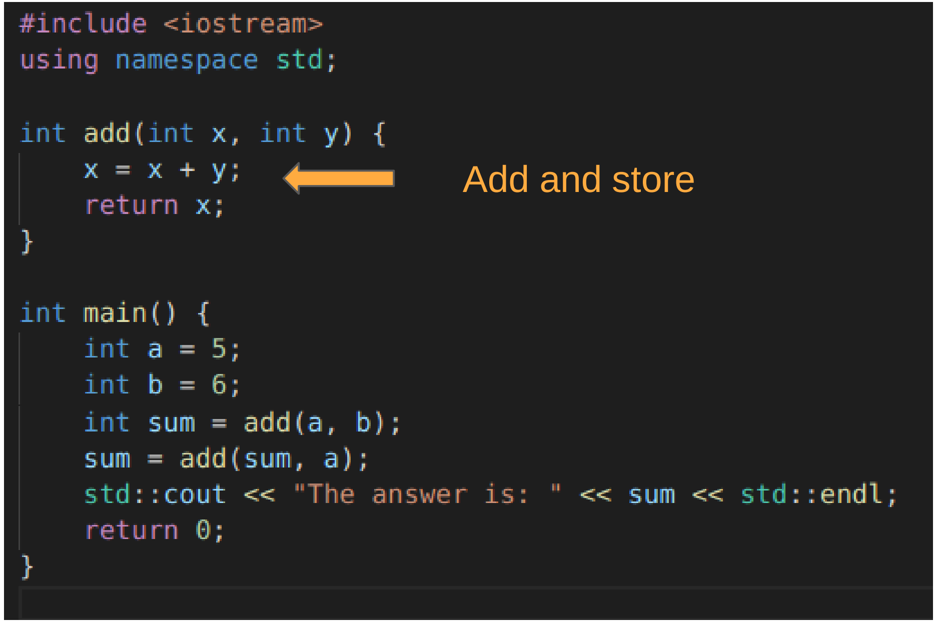
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Copy Values

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	~~?~~ 11
int	x	**~~?~~ ~~5~~ ~~11~~ 11**
int	y	**~~?~~ ~~5~~ 5**

Example 1: Pass by Value



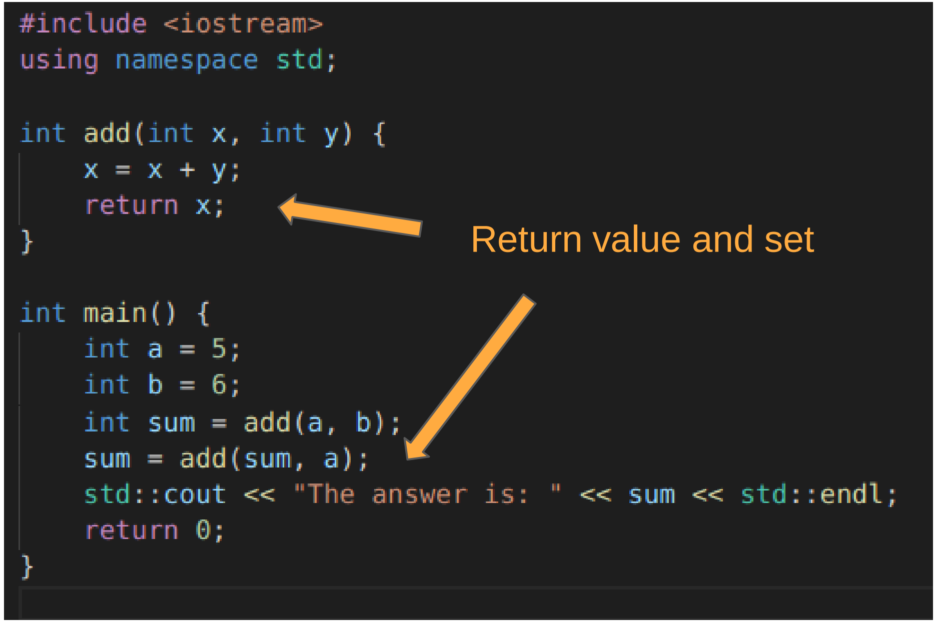
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Add and store

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	~~?~~ 11
int	x	**~~?~~ ~~5~~ ~~11~~ ~~11~~ 16**
int	y	~~?~~ ~~5~~ 5

Example 1: Pass by Value



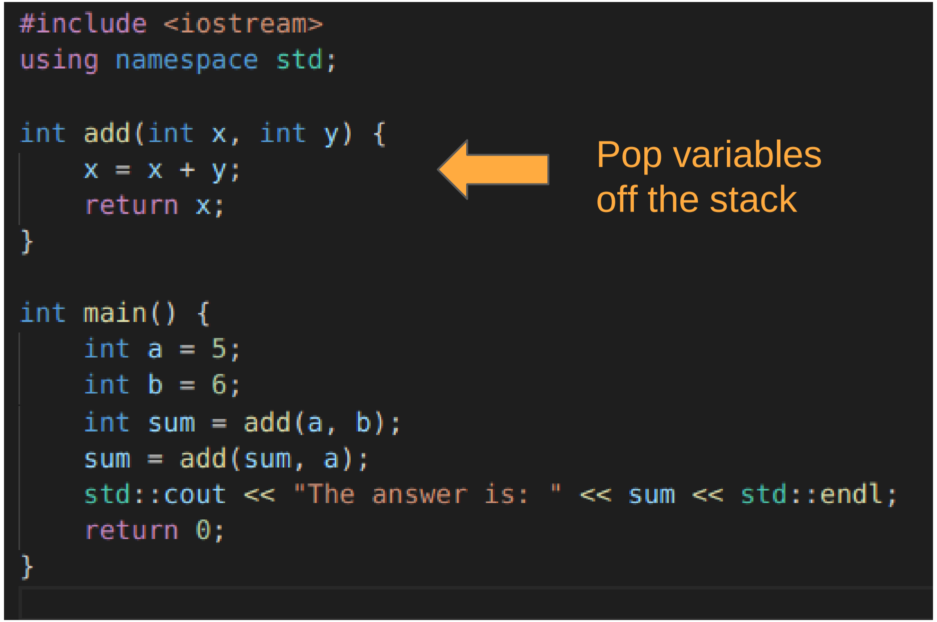
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Return value and set

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	**~~?~~ ~~11~~ 16**
int	x	~~?~~ ~~5~~ ~~11~~ ~~11~~ 16
int	y	~~?~~ ~~5~~ 5

Example 1: Pass by Value



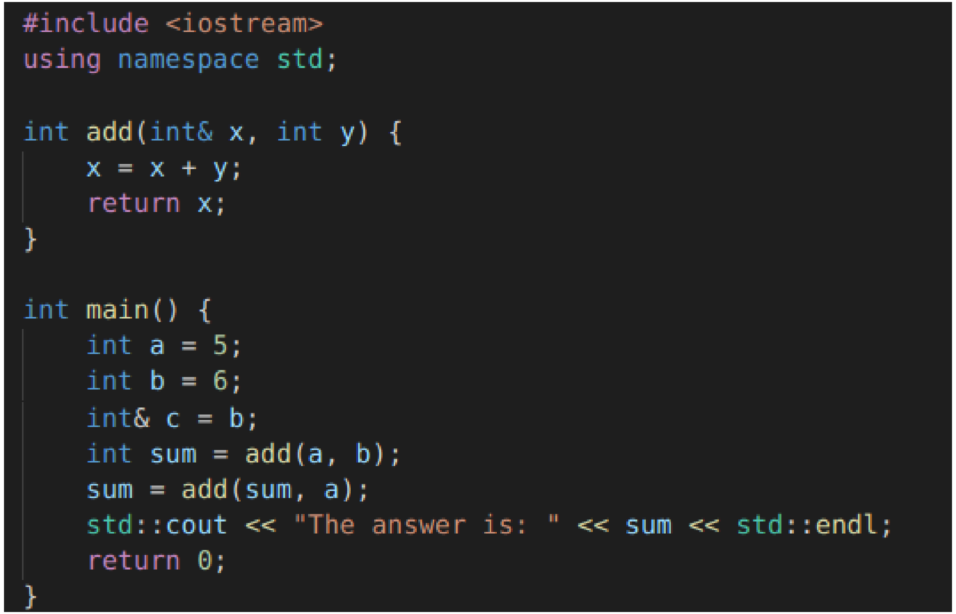
```
#include <iostream>
using namespace std;

int add(int x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Pop variables off the stack

Memory

Type	Name	Value
int	a	~~?~~ 5
int	b	~~?~~ 6
int	sum	~~?~~ ~~11~~ 16
		~~?~~ ~~5~~ ~~11~~ ~~11~~ 16
		~~?~~ ~~5~~ 5

Example 2: Pass by Reference

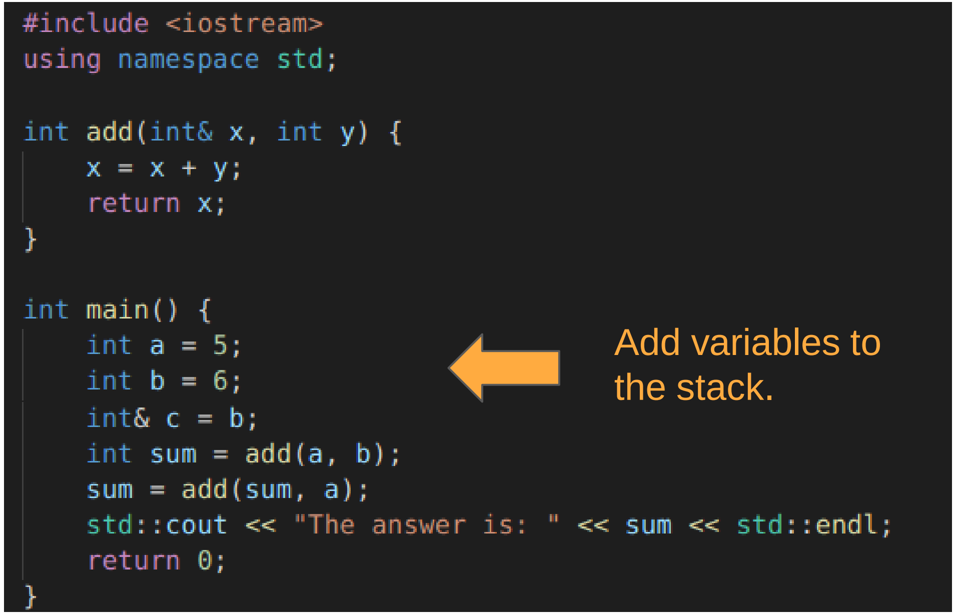


```cpp
#include <iostream>
using namespace std;

int add(int& x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int& c = b;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

Memory

Type	Name	Value

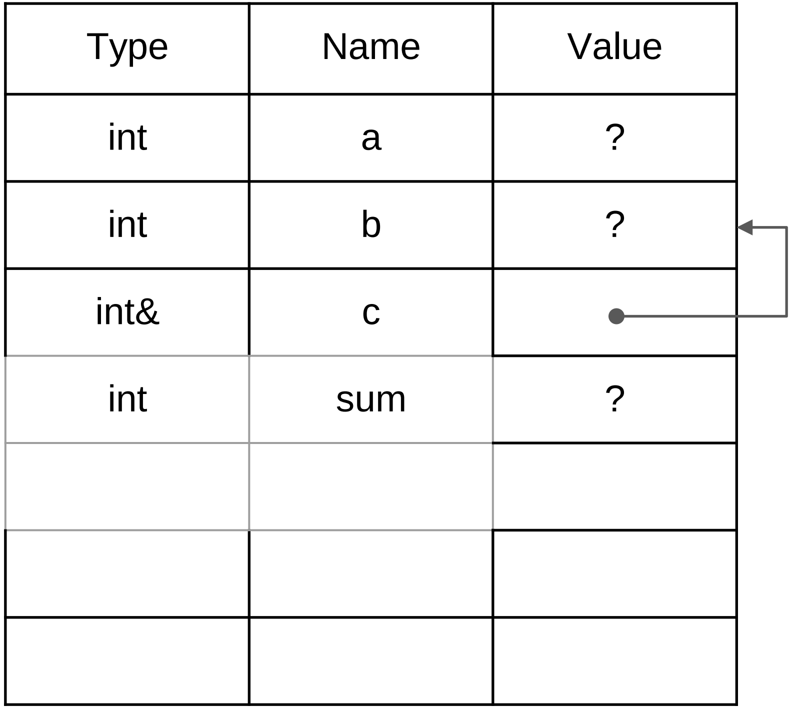
Example 2: Pass by Reference



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

int add(int& x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int& c = b;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

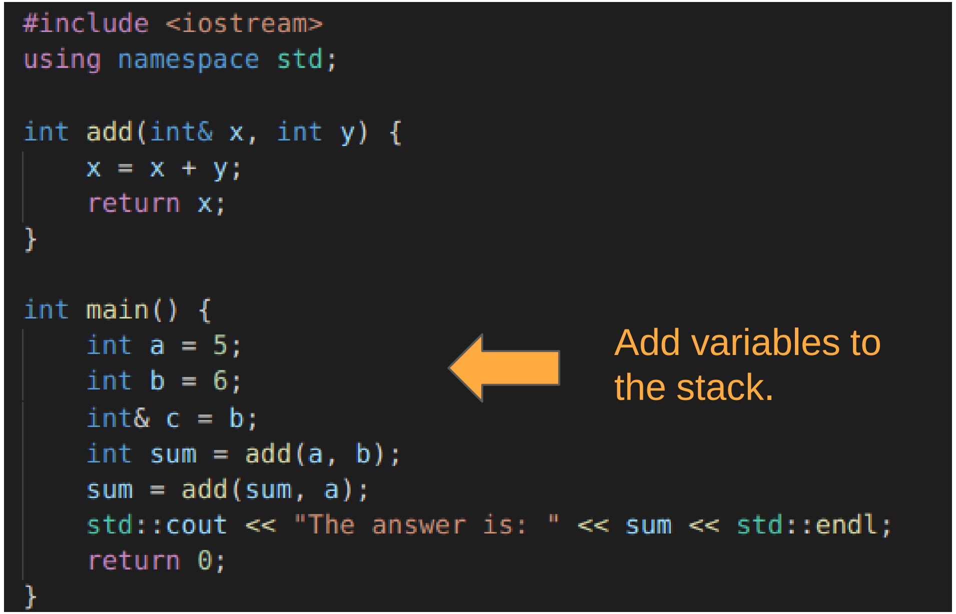
Add variables to the stack.

Memory



Type	Name	Value
int	a	?
int	b	?
int&	c	●
int	sum	?

Example 2: Pass by Reference



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

int add(int& x, int y) {
    x = x + y;
    return x;
}

int main() {
    int a = 5;
    int b = 6;
    int& c = b;
    int sum = add(a, b);
    sum = add(sum, a);
    std::cout << "The answer is: " << sum << std::endl;
    return 0;
}
```

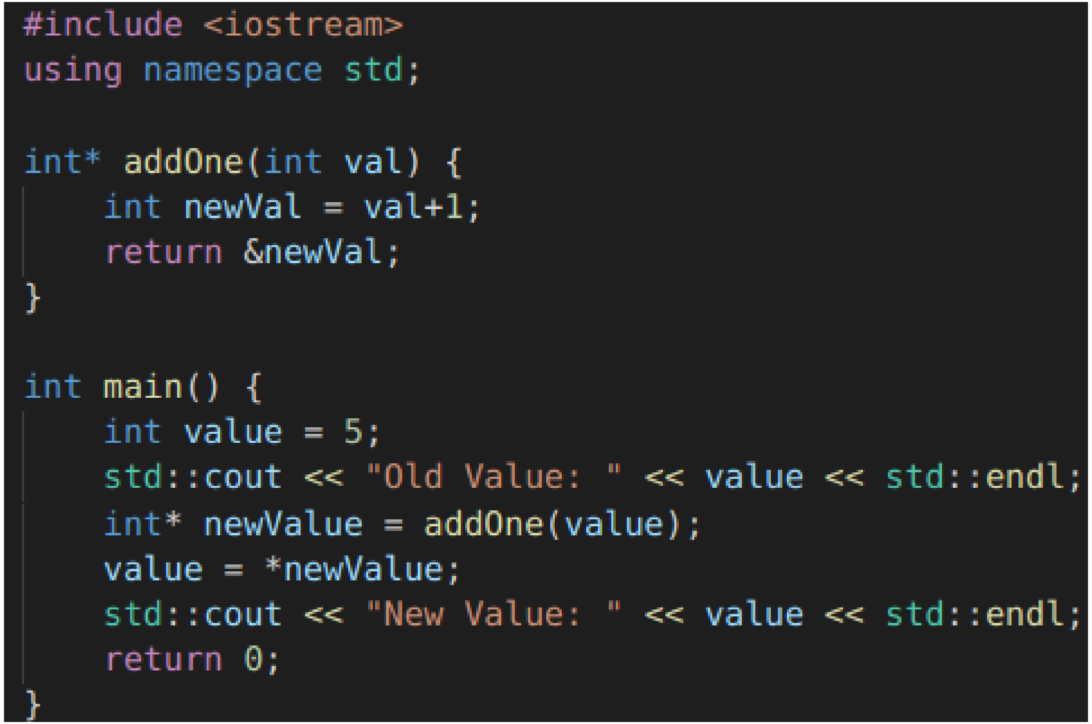
Add variables to the stack.

Complete Together In Class

Memory

Type	Name	Value
int	a	?
int	b	?
int&	c	●
int	sum	?

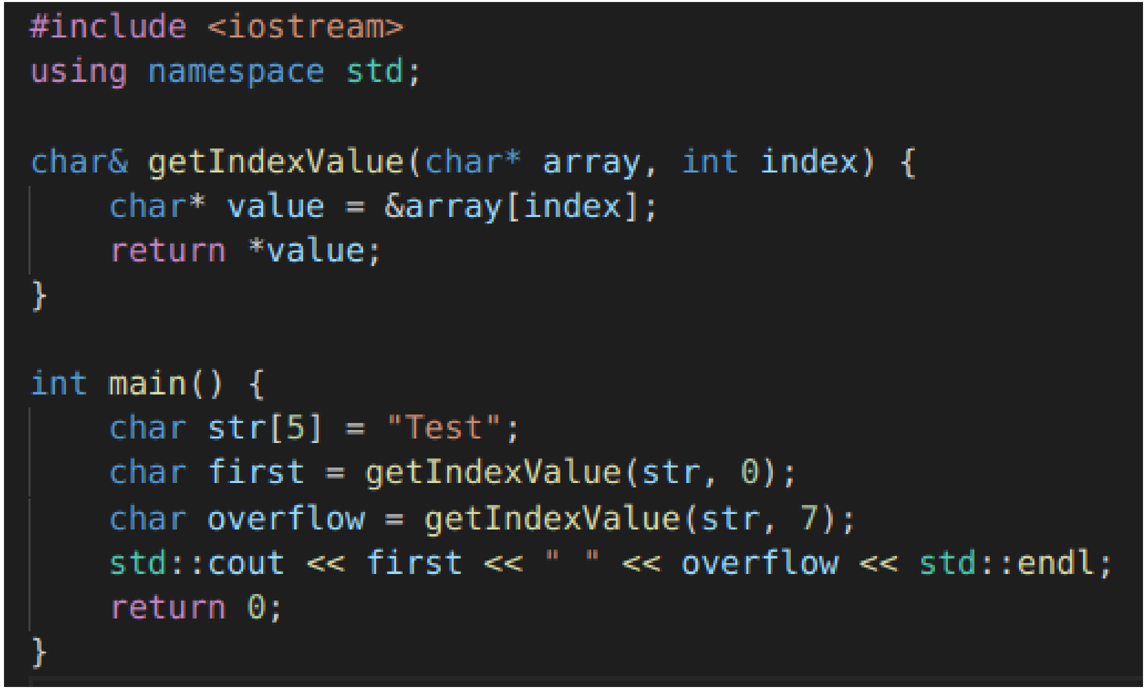
Example 3: Segmentation Fault



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

int* addOne(int val) {
    int newVal = val+1;
    return &newVal;
}

int main() {
    int value = 5;
    std::cout << "Old Value: " << value << std::endl;
    int* newValue = addOne(value);
    value = *newValue;
    std::cout << "New Value: " << value << std::endl;
    return 0;
}
```

Complete Together In Class

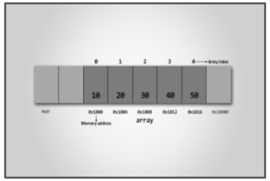
Example 4: Buffer Overflow



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

char& getIndexValue(char* array, int index) {
    char* value = &array[index];
    return *value;
}

int main() {
    char str[5] = "Test";
    char first = getIndexValue(str, 0);
    char overflow = getIndexValue(str, 7);
    std::cout << first << " " << overflow << std::endl;
    return 0;
}
```

Try on your own.

Roadmap for Today



Practice: Diagramming Memory

Dynamic Memory

Classes and Dynamic Memory

What is conceptually wrong with the following application?

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

int main()
{
            string employees[50];
            int numEmployees = 0;

            employees[0] = "Alice";
      numEmployees++;
            employees[1] = "Bob";
      numEmployees++;
            employees[2] = "Beth";
      numEmployees++;

      for (int i = 0; i < numEmployes; i++) {
            cout << employees[i] << endl;
      }
}
```

What is conceptually wrong with the following application?

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

int main()
{
            string employees[50];
            int numEmployees = 0;

            employees[0] = "Alice";
      numEmployees++;
            employees[1] = "Bob";
      numEmployees++;
            employees[2] = "Beth";
      numEmployees++;

      for (int i = 0; i < numEmployes; i++) {
            cout << employees[i] << endl;
      }
}
```

We have to specify the size of the array!

- What if we have more than 50 employees?
- Increasing the size will cause a recompile.

What is conceptually wrong with the following application?

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

int main()
{
              int size;
              cin >> size;

              string employees[size];
              int numEmployees = 0;

              employees[0] = "Alice";
       numEmployees++;
              employees[1] = "Bob";
       numEmployees++;
              employees[2] = "Beth";
       numEmployees++;

       for (int i = 0; i < numEmployes; i++) {
              cout << employees[i] << endl;
       }
       delete[] employees;
}
```

We have to specify the size of the array!

- What if we have more than 50 employees?
- Increasing the size will cause a recompile.
- Dynamic array sizes are not possible on the stack.

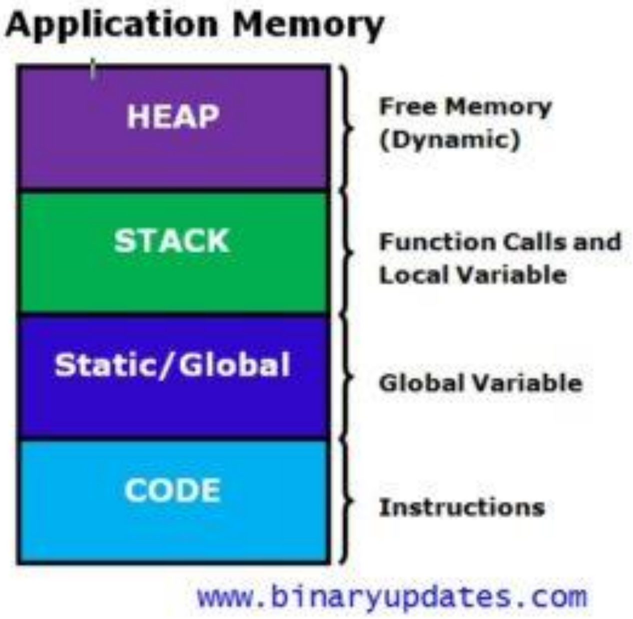
Solution: Use dynamic memory allocation (new / delete)

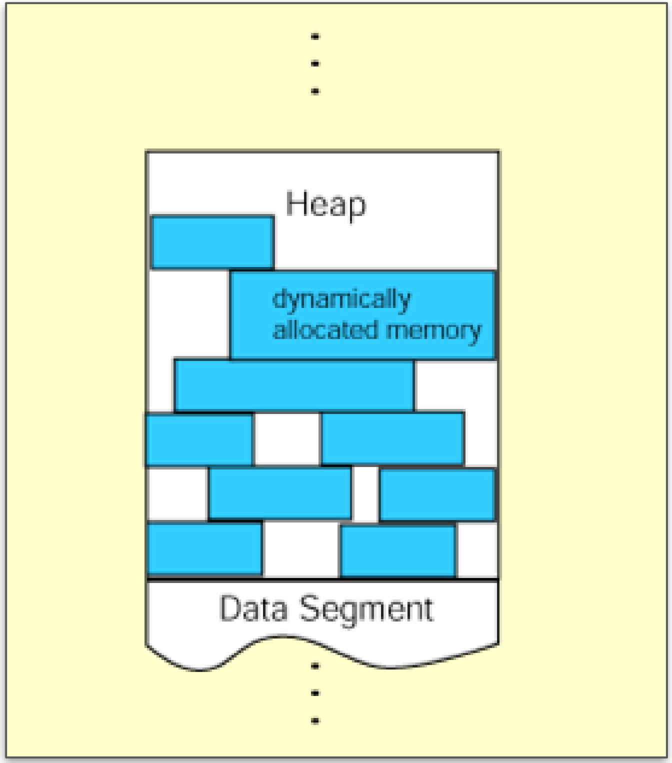
```cpp
#include <iostream>
using namespace std;

int main()
{
              int size;
              cin >> size;

              string* employees = new string[size];
              int numEmployees = 0;

              employees[0] = "Alice";
       numEmployees++;
              employees[1] = "Bob";
       numEmployees++;
              employees[2] = "Beth";
       numEmployees++;

       for (int i = 0; i < numEmployes; i++) {
              cout << employees[i] << endl;
       }
       delete[] employees;
}
```

Creates a dynamic array based on input.

Deletes the dynamic array.

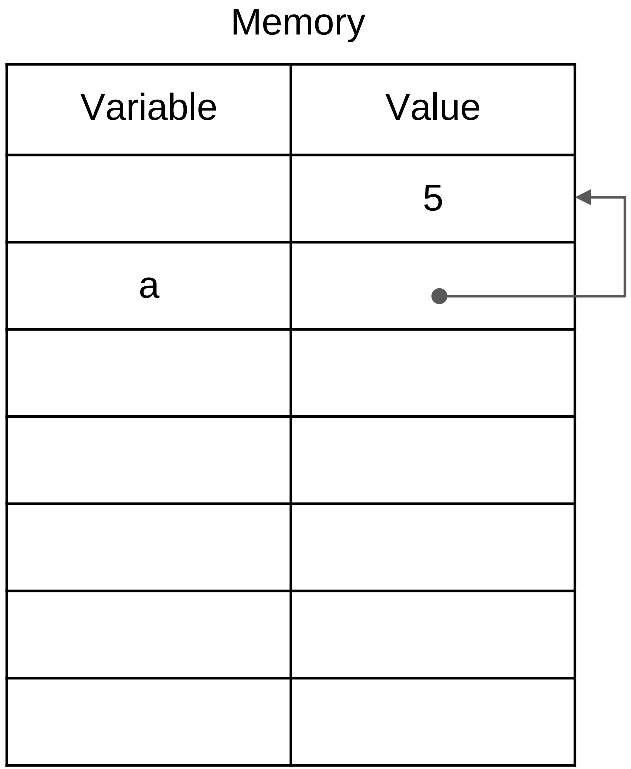
Dynamic memory allocation occurs on the **heap** instead of the **stack**.





The new operator creates memory on the **heap** instead of the **stack**.

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

int main()
{
        int* a = new int(5);
        cout << *a << endl;
}
```

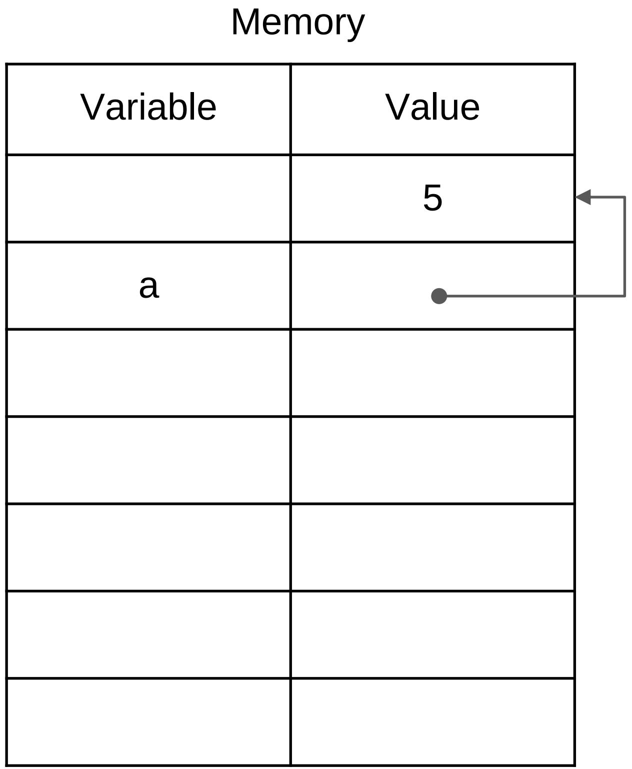


Memory

Variable	Value
	5
a	

The new operator creates memory on the **heap** instead of the **stack**.

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

int main()
{
        int* a = new int(5);
        cout << *a << endl;
}
```

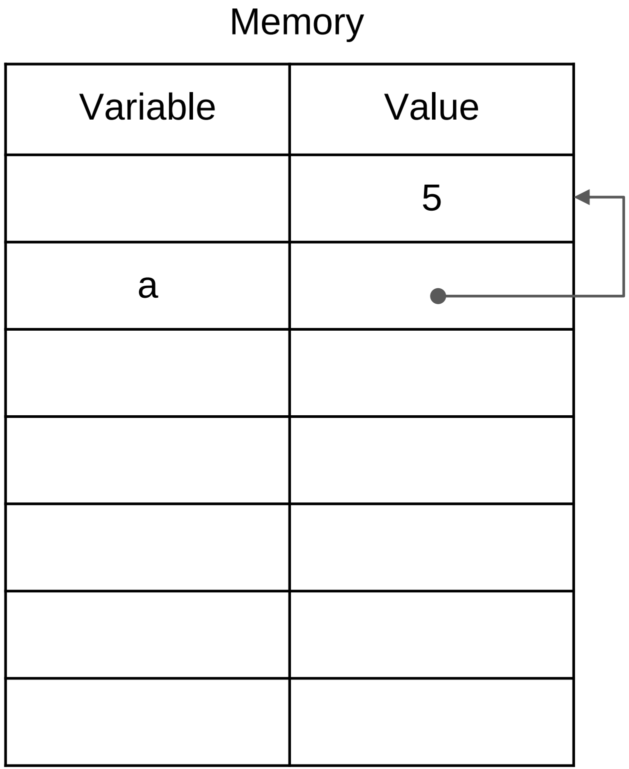


Memory

Variable	Value
	5
a	

We must always delete the allocated memory off the heap with the delete keyword.

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

int main()
{
        int* a = new int(5);
        cout << *a << endl;
        delete a;
}
```

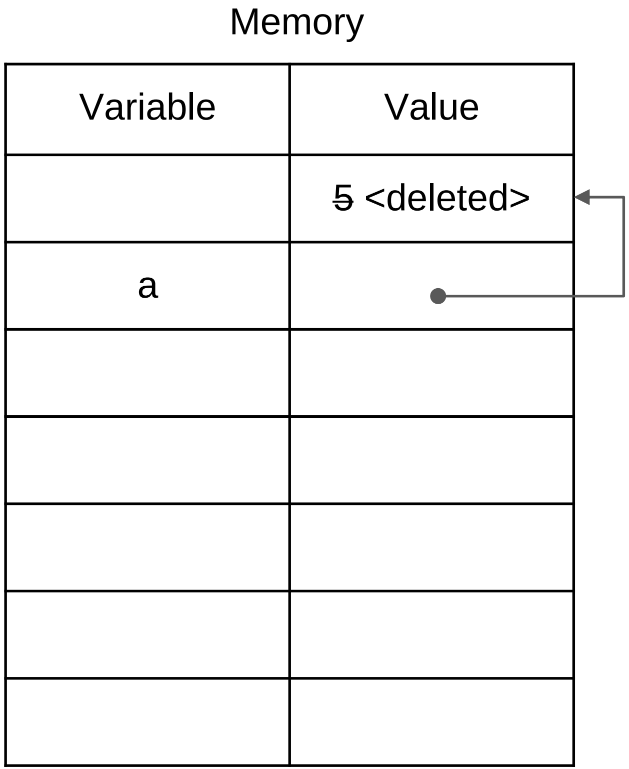


Memory

Variable	Value
	5
a	●

We must always delete the allocated memory off the heap with the delete keyword.

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

int main()
{
        int* a = new int(5);
        cout << *a << endl;
        delete a;
}
```



Memory

Variable	Value
	~~5~~ <deleted>
a	

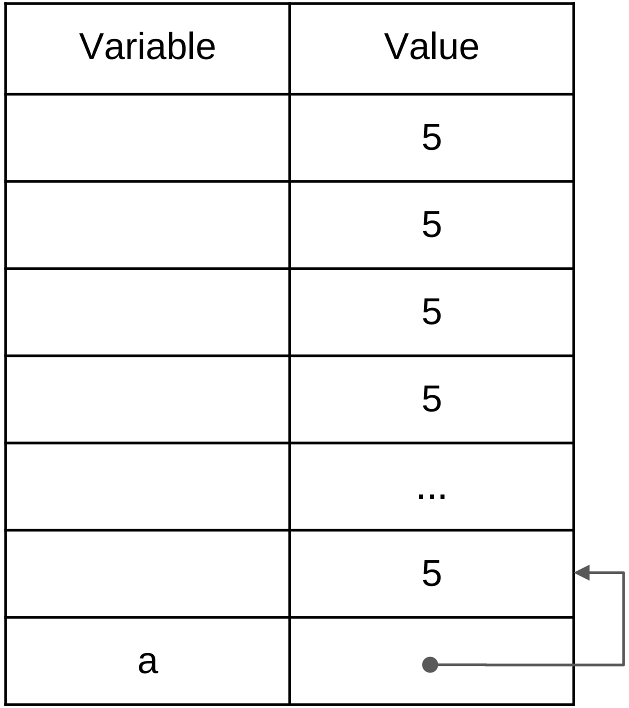
Let’s look at an example. What is wrong with this program?

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

int main()
{
         int* a;
         while (true) {
         a = new int(5);
                  cout << *a <<
endl;
         }
         delete a;
}
```

A memory leak is caused when we don’t delete our allocated memory off the heap.

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

int main()
{
        int* a;
        while (true) {
        a = new int(5);
                cout << *a <<
endl;
        }
        delete a;
}
```

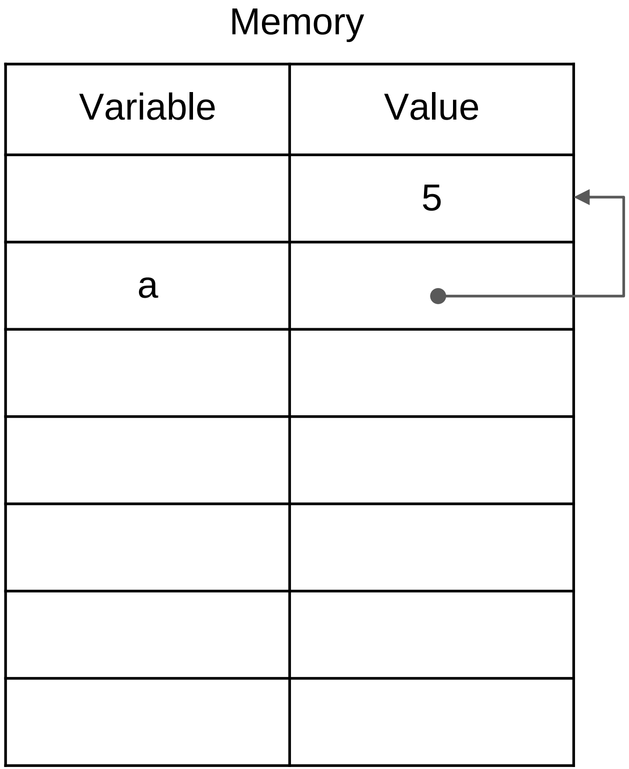
Memory



Variable	Value
	5
	5
	5
	5
	...
	5
a	●

We can fix the memory leak by deleting every value we allocate.

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

int main()
{
        int* a;
        while (true) {
        a = new int(5);
                cout << *a <<
endl;
                delete a;
        }
        // delete a;
}
```

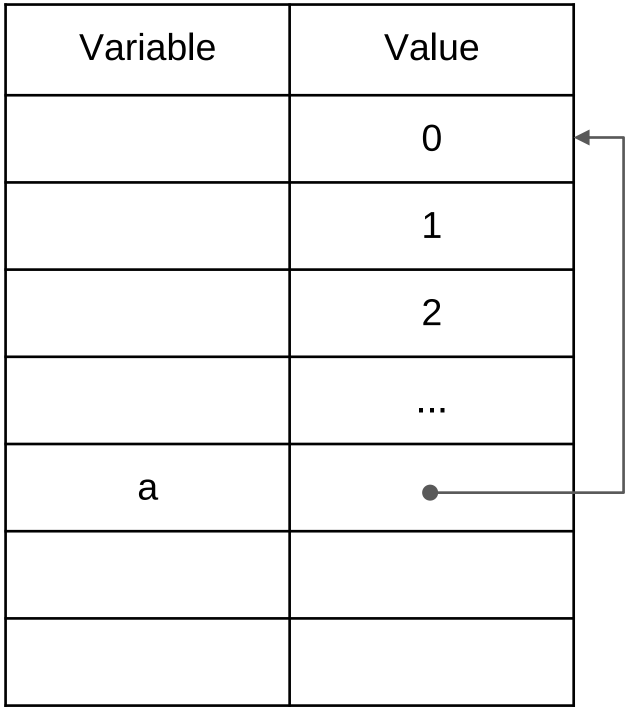


Memory

Variable	Value
	5
a	●

To create and destroy a dynamic **array** on the heap, we can use new type[] and delete[].

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

int main()
{
         int dynamicSize;
         cin >> dynamicSize;
         int* a = new
int[dynamicSize];
         a[0] = 0;
         a[1] = 1;
         a[2] = 2;
         delete[] a;
}
```



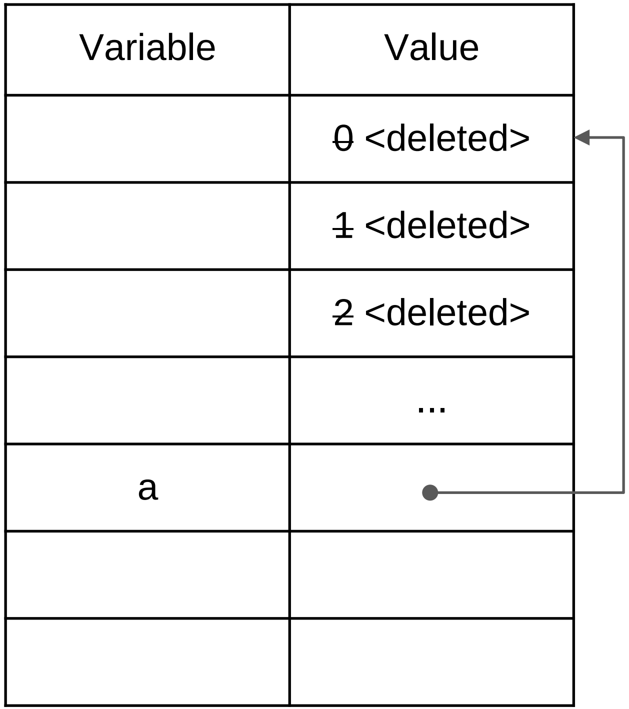
Memory

Variable	Value
	0
	1
	2
	...
a	●

To create and destroy a dynamic **array** on the heap, we can use new type[] and delete[].

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

int main()
{
        int dynamicSize;
        cin >> dynamicSize;
        int* a = new
int[dynamicSize];
        a[0] = 0;
        a[1] = 1;
        a[2] = 2;
        delete[] a;
}
```

Memory

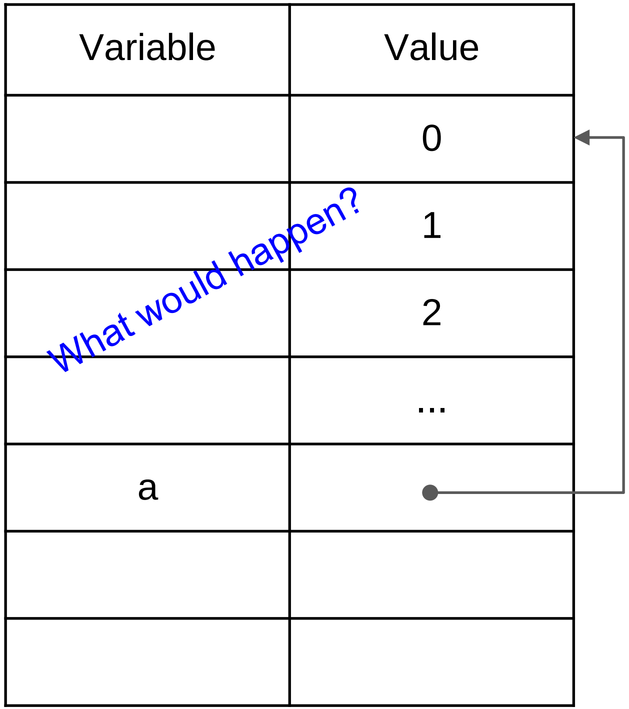


Variable	Value
	~~0~~ <deleted>
	~~1~~ <deleted>
	~~2~~ <deleted>
	...
a	●

If we just used delete, it would not deallocate the array.

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

int main()
{
        int dynamicSize;
        cin >> dynamicSize;
        int* a = new
int[dynamicSize];
        a[0] = 0;
        a[1] = 1;
        a[2] = 2;
        // delete[] a;
        delete a;
```

Memory

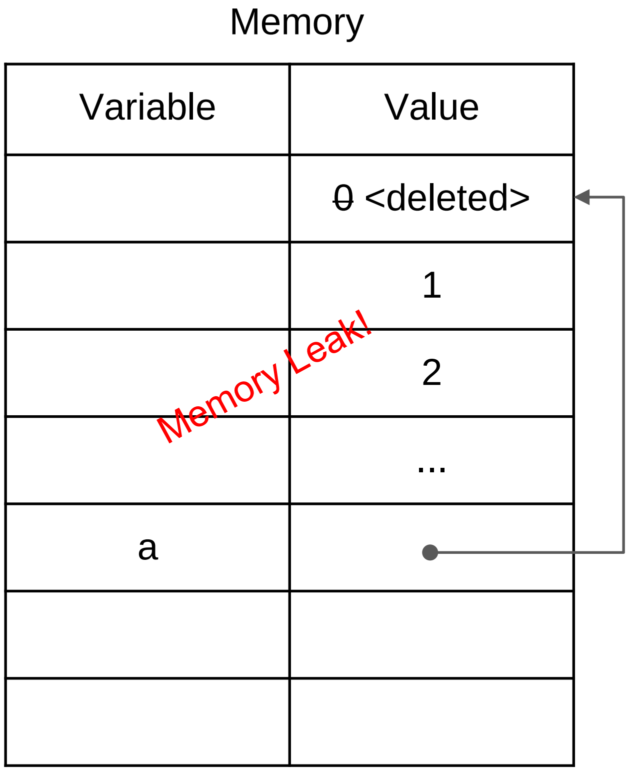


Variable	Value
	0
	1
	2
	...
a	●

What would happen?

If we just used delete, it would not deallocate the array.

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

int main()
{
          int dynamicSize;
          cin >> dynamicSize;
          int* a = new
int[dynamicSize];
          a[0] = 0;
          a[1] = 1;
          a[2] = 2;
          // delete[] a;
          delete a;
```



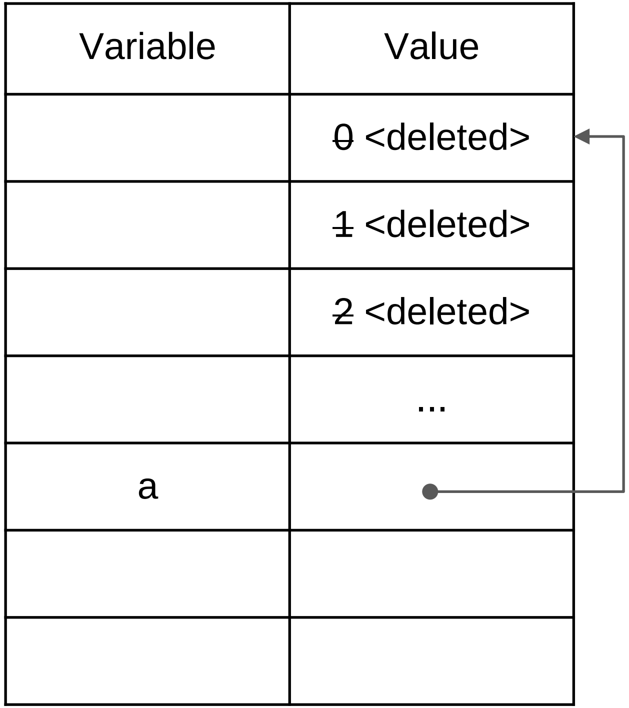
Memory

Variable	Value
	~~0~~ <deleted>
	1
	2
	...
a	●

To create and destroy a dynamic **array** on the heap, we can use new type[] and delete[].

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

int main()
{
        int dynamicSize;
        cin >> dynamicSize;
        int* a = new
int[dynamicSize];
        a[0] = 0;
        a[1] = 1;
        a[2] = 2;
        delete[] a;
}
```

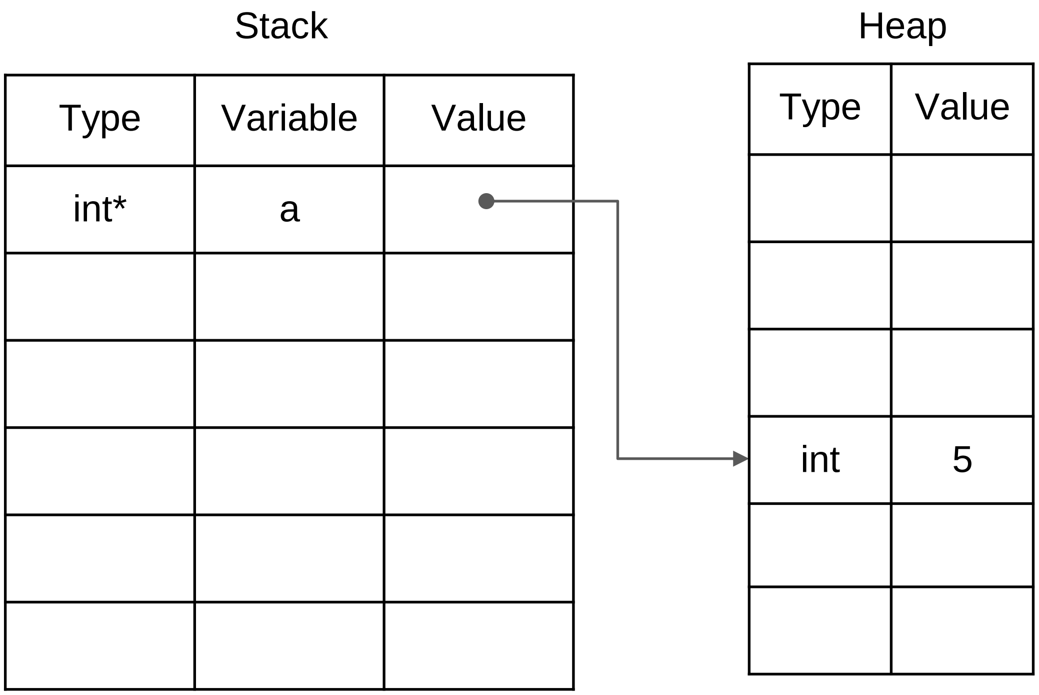
Memory



Variable	Value
	~~0~~ <deleted>
	~~1~~ <deleted>
	~~2~~ <deleted>
	...
a	●

Stack vs. Heap: new and delete allocates memory on the heap

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

int main()
{
         int* a = new
int(5);
         cout << *a <<
endl;
         delete a;
}
```



Stack

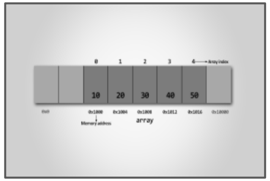
Type	Variable	Value
int*	a	●

Heap

Type	Value
int	5

This is a picture of how the stack and heap are related.

Roadmap for Today



Practice: Diagramming Memory

Dynamic Memory

Classes and Dynamic Memory

What is wrong with the following code?

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            double& operator[](int index) {
                        return array[index];
            }

private:
            int size;
            double* array;
};


int main() {
            VectorXD vec(5);
            vec[0] = 5;

            return 0;
}
```

What is wrong with the following code?

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            double& operator[](int index) {
                        return array[index];
            }

private:
            int size;
            double* array;
};


int main() {
            VectorXD vec(5);
            vec[0] = 5;

            return 0;
}
```

Memory Leak!
(array is never deleted)

How can we fix this?

Let’s follow good memory management practices.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            ~VectorXD() {
                        delete[]
            }

            double& operator[](int index) {
                        return array[index];
            }

private:
            int size;
            double* array;
};


int main() {
            VectorXD vec(5);
            vec[0] = 5;

            return 0;
}
```

Add Destructor
(called when variable goes out of scope)

Let’s follow good memory management practices.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            ~VectorXD() {
                        delete[] array;
            }

            double& operator[](int index) {
                        return array[index];
            }

private:
            int size;
            double* array;
};


int main() {
            VectorXD vec(5);
            vec[0] = 5;
            VectorXD vec2(vec)
            vec = vec2;
            return 0;
}
```

Now what is wrong?

Let’s follow good memory management practices.

```
class VectorXD {
public:
        VectorXD(int size) : size(size) {
                array = new double[size];
        }

        ~VectorXD() {
                delete[] array;
        }

        double& operator[](int index) {
                return array[index];
        }

private:
        int size;
        double* array;
};


int main() {
        VectorXD vec(5);
        vec[0] = 5;
        VectorXD vec2(vec)
        vec = vec2;
        return 0;
}
```

Copy Constructor or **operator=** will do a direct copy of The data!

Changing one vector will change the data of another.

How can we fix this?

Let’s follow good memory management practices.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            VectorXD(const VectorXD& vec) {
                        array = NULL;
                        *this = vec;
            }

      ~VectorXD() {
                        delete[] array;
            }

            void operator=(const VectorXD& ve
            this->size = vec.size;
            delete[] this->array;
            this->array = new double[this->size];
            for (int i = 0; i < size; i++) {
                        this->array[i] =
      vec.array[i];
            }
            }
private:
            int size;
            double* array;
};
```

Add Copy Constructor

Overload assignment operator

Let’s follow good memory management practices.

```
class VectorXD {
public:
          VectorXD(int size) : size(size) {
                    array = new double[size];
          }

          VectorXD(const VectorXD& vec) {
                    array = NULL;
                    *this = vec;
          }

     ~VectorXD() {
                    delete[] array;
          }

          void operator=(const VectorXD& ve
          this->size = vec.size;
          delete[] this->array;
          this->array = new double[this->size];
          for (int i = 0; i < size; i++) {
                    this->array[i] =
     vec.array[i];
          }
          }
private:
          int size;
          double* array;
};
```

Add Copy Constructor

Notice the copy constructor can call the assignment operator.

Overload assignment operator

The Big Three: Anytime a class uses dynamic memory, remember to implement these.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            VectorXD(const VectorXD& vec) {
                        array = NULL;
                        *this = vec;
            }

      ~VectorXD() {
                        delete[] array;
            }

            void operator=(const VectorXD& vec) {
            this->size = vec.size;
            delete[] this->array;
            this->array = new double[this->size];
            for (int i = 0; i < size; i++) {
                        this->array[i] =
      vec.array[i];
            }
            }
private:
            int size;
            double* array;
};
```

The Big Three

Copy Constructor

Destructor

Assignment operator

The Big Three: Anytime a class uses dynamic memory, remember to implement these.

```
class VectorXD {
public:
      VectorXD(int size) : size(size) {
            array = new double[size];
      }

      VectorXD(const VectorXD& vec) {
            array = NULL;
            *this = vec;
      }

   ~VectorXD() {
            delete[] array;
      }

      void operator=(const VectorXD& vec) {
      this->size = vec.size;
      delete[] this->array;
      this->array = new double[this->size];
      for (int i = 0; i < size; i++) {
            this->array[i] =
   vec.array[i];
      }
      }
private:
      int size;
      double* array;
};
```

It is often error prone to remember when you need to delete an array or pointer.

Causes a memory leak if you forget!

The Big Three: Anytime a class uses dynamic memory, remember to implement these.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            VectorXD(const VectorXD& vec) {
                        array = NULL;
                        *this = vec;
            }

      ~VectorXD() {
                        ~~delete[] array;~~
            }

            void operator=(const VectorXD& vec) {
            this->size = vec.size;
            ~~delete[] this->array;~~
            this->array = new double[this->size];
            for (int i = 0; i < size; i++) {
                        this->array[i] =
      vec.array[i];
            }
            }
private:
            int size;
            double* array;
};
```

It is often error prone to remember when you need to delete an array or pointer.

Causes a memory leak if you forget!

Wouldn't it be nice to just say ***new*** *like Java and not worry about the delete.*

However, no garbage collection.

The Big Three: Anytime a class uses dynamic memory, remember to implement these.

```
class VectorXD {
public:
            VectorXD(int size) : size(size) {
                        array = new double[size];
            }

            VectorXD(const VectorXD& vec) {
                        array = NULL;
                        *this = vec;
            }

      ~VectorXD() {
                        ~~delete[] array;~~
            }

            void operator=(const VectorXD& vec) {
            this->size = vec.size;
            ~~delete[] this->array;~~
            this->array = new double[this->size];
            for (int i = 0; i < size; i++) {
                        this->array[i] =
      vec.array[i];
            }
            }
private:
            int size;
            double* array;
};
```

It is often error prone to remember when you need to delete an array or pointer.

Causes a memory leak if you forget!

Wouldn't it be nice to just say **new** like Java and not worry about the delete.

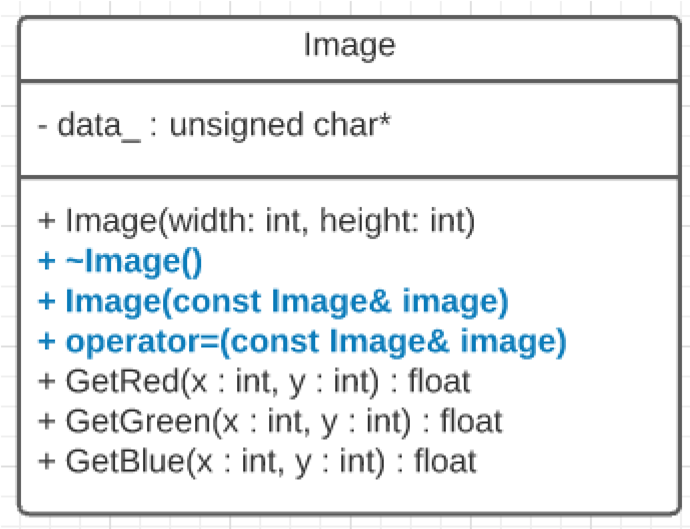
However, no garbage collection.
There is a solution!

Whenever we work with dynamic memory in classes, we need to think about the **Big Three**.

The Big Three:

- Destructor
- Copy Constructor
- Assignment Operator

Example:



Image

- data_ : unsigned char*

+ Image(width: int, height: int)
+ ~Image()
+ Image(const Image& image)
+ operator=(const Image& image)
+ GetRed(x : int, y : int) : float
+ GetGreen(x : int, y : int) : float
+ GetBlue(x : int, y : int) : float

Whenever we work with dynamic memory in classes, we need to think about the **Big Three**.

The Big Three:

- Destructor
- Copy Constructor
- Assignment Operator

```
Image::Image(int width, int height) {
            data_ = new unsigned char[width*height*3];
}

Image::~Image() {
            delete[] data_;
}

Image::Image(const Image& image) {
            *this = image;
}

void Image::operator=(const Image& image) {
      delete[] data_;
            data_ = new unsigned char[image.GetSize()];
            memcpy(data_, image.data_, image.GetSize());
}
```

Calls assignment operator.

Example:

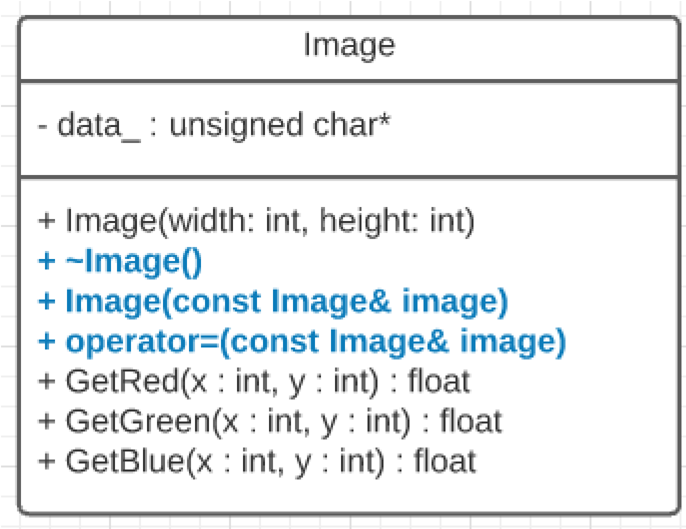


Image
- data_ : unsigned char*
+ Image(width: int, height: int)
+ ~Image()
+ Image(const Image& image)
+ operator=(const Image& image)
+ GetRed(x : int, y : int) : float
+ GetGreen(x : int, y : int) : float
+ GetBlue(x : int, y : int) : float

Whenever we work with dynamic memory in classes, we need to think about the **Big Three**.

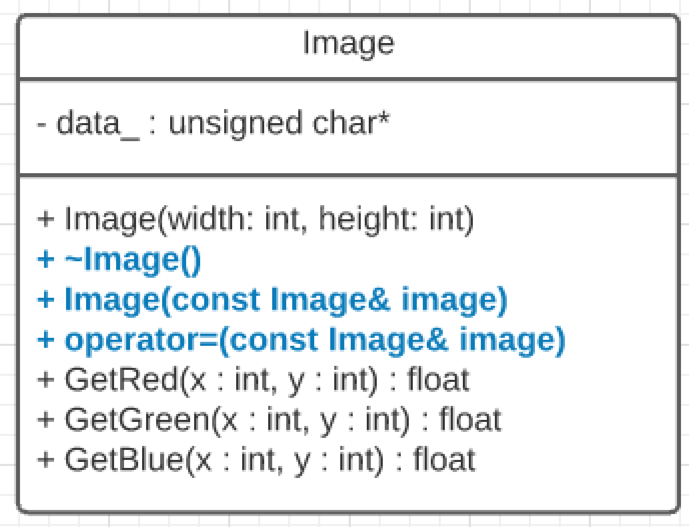
The Big Three:

- Destructor
- Copy Constructor
- Assignment Operator

```
Image::Image(int width, int height) {
            data_ = new unsigned char[width*height*3];
}

Image::~Image() {
            delete[] data_;
}

Image::Image(const Image& image) {
            *this = image;
}

void Image::operator=(const Image& image) {
      delete[] data_;
            data_ = new unsigned char[image.GetSize()];
            memcpy(data_, image.data_, image.GetSize());
}
```

Calls assignment operator.

Example:



Image

- data_ : unsigned char*

+ Image(width: int, height: int)
+ ~Image()
+ Image(const Image& image)
+ operator=(const Image& image)
+ GetRed(x : int, y : int) : float
+ GetGreen(x : int, y : int) : float
+ GetBlue(x : int, y : int) : float

this keyword is a pointer to the current class.

***this** dereferences the pointer to get the value.

Summary

`* & ->`

Pointers

- Pointers are memory addresses
- The can point to any location in memory.
- The memory doesn’t need to exist (e.g. NULL).

`new / delete`

Dynamic Memory

- Anytime a **new** is used on the heap, there must be a **delete.**
- If you allocate with **new[]** you must delete memory with **delete[].**
- The **Big Three** (destructor, copy constructor, assignment operator)