CS 105C: Lecture 2

Questions!

Q: Why is it that we sometimes use the :: operator and sometimes use the . operator?

```
1  class Dog { ... };
2
3  Dog d1;
4
5  // We use the :: operator to access
6  // things belonging to the Dog class
7  // or another namespace
8
9  Dog::standardNumTeeth;
10
11 // We use the . operator to access
12 // a thing belonging to an object
13
14 d1.numTeeth;
```

Questions!

Q: Why can compiler optimization change the meaning of a program when pipeline optimization can not?

A: In a legal sense, compiler optimization just exposes a broken program as broken.

A: Pipeline optimization can also break incorrect assembly programs! (In particular, multithreaded programs that do not properly use synchronization)

CS 105C: Lecture 2

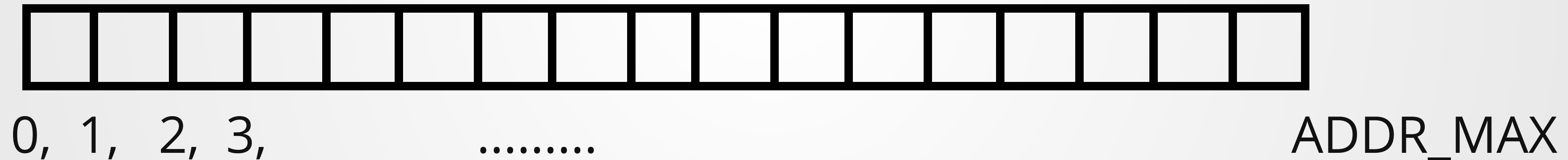
Pointers, References, and Classes

Basics of Memory

Memory in a C++ Program

One giant array

Each element is one byte large

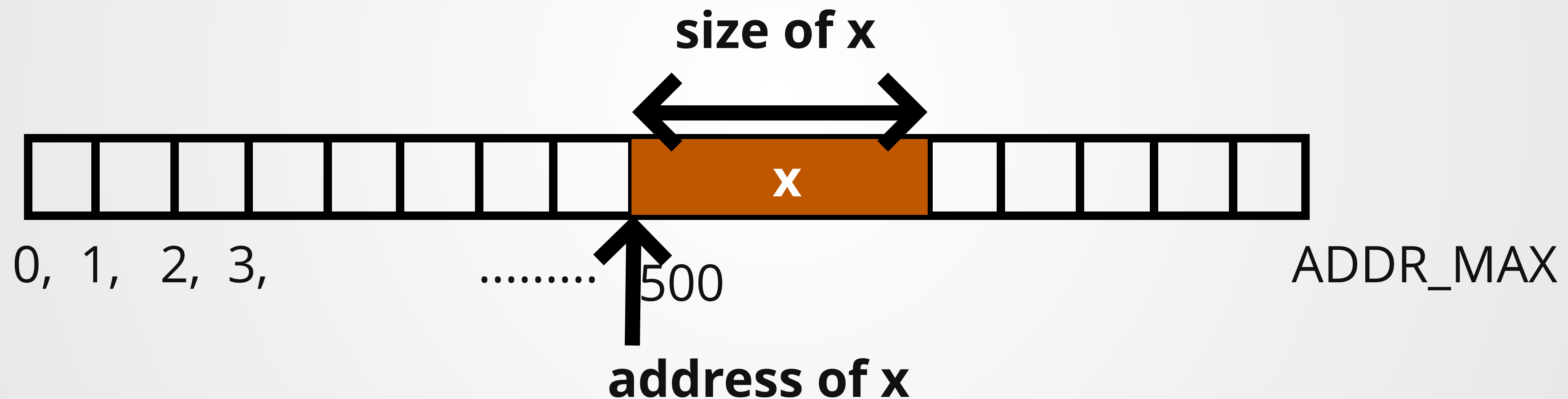


Numbering starts at 0 and runs to the size of the address space.

Memory in a C++ Program

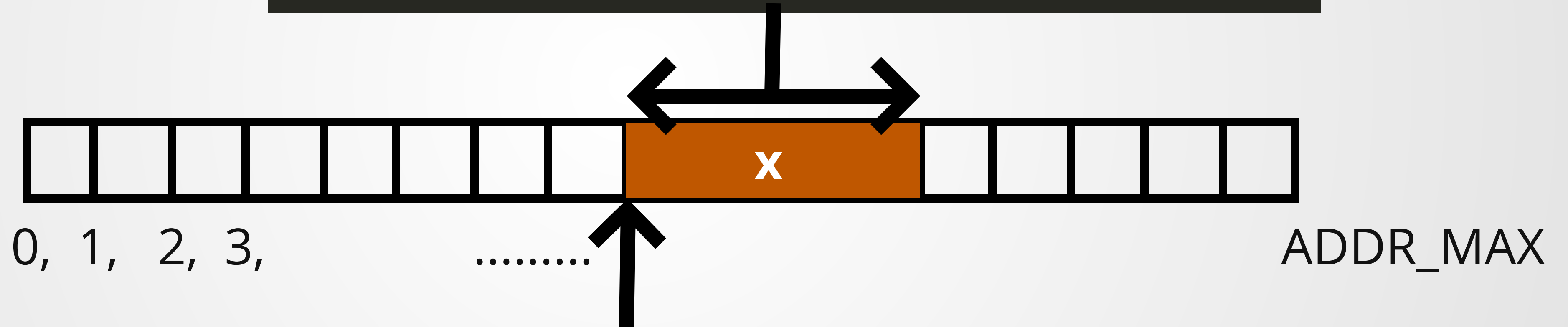
Creating a variable uses up some of this memory

For most variables, this takes place at variable declaration



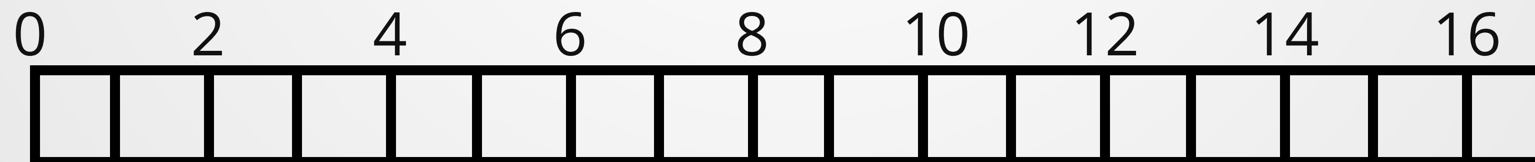
We say that the index of the first byte is the *address* of the variable, and the number of bytes it takes up is its *size*

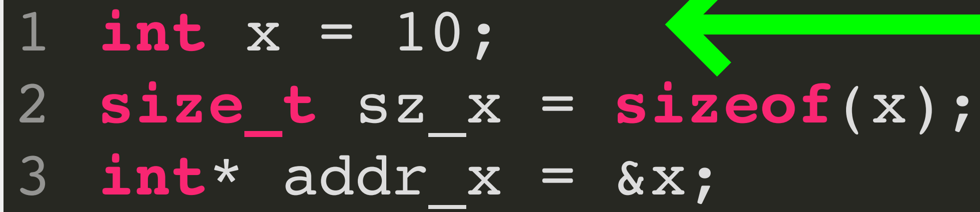
Memory in a C++ Program



```c
int x = 10;
size_t sz_x = sizeof(x);
int* addr_x = &x;
```

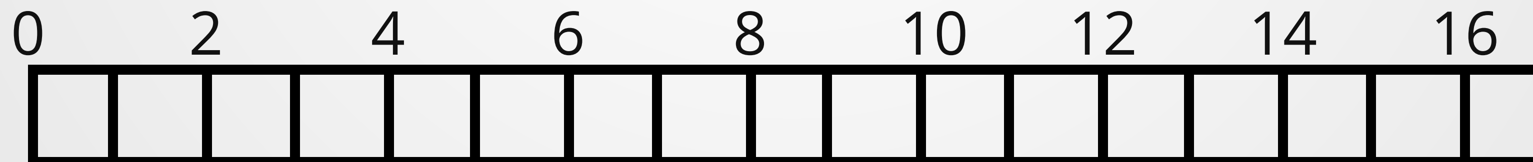
Name	Type	Address





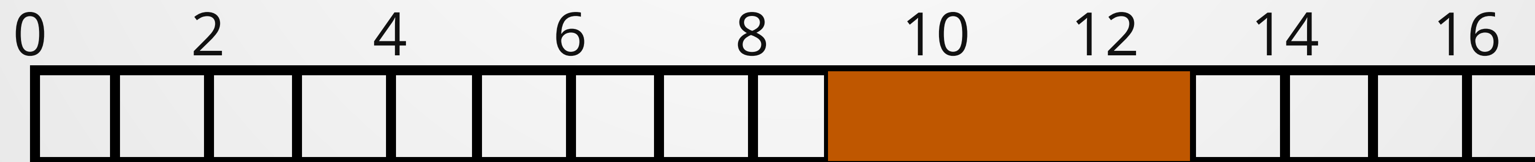
```
1 int x = 10;
2 size_t sz_x = sizeof(x);
3 int* addr_x = &x;
```

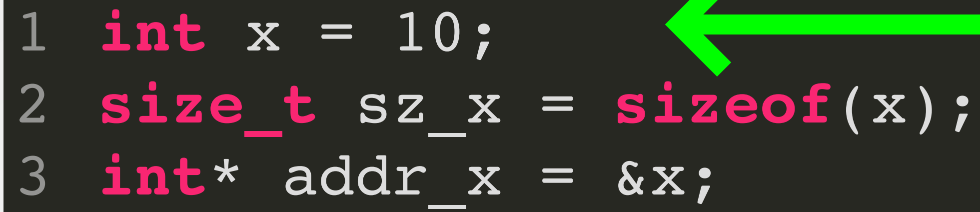
Name	Type	Address



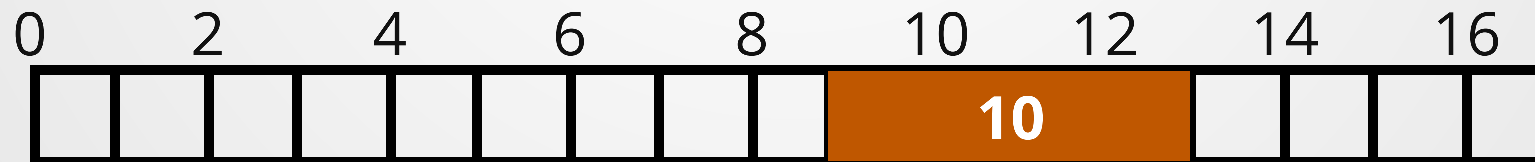
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

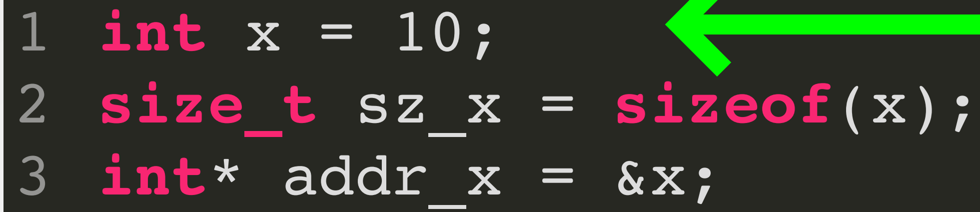
Name	Type	Address





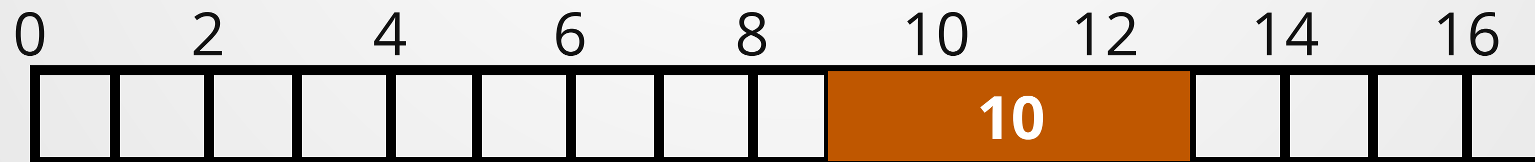
Name	Type	Address





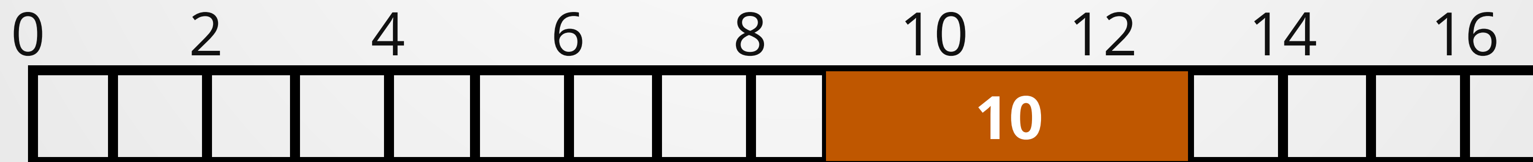
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9



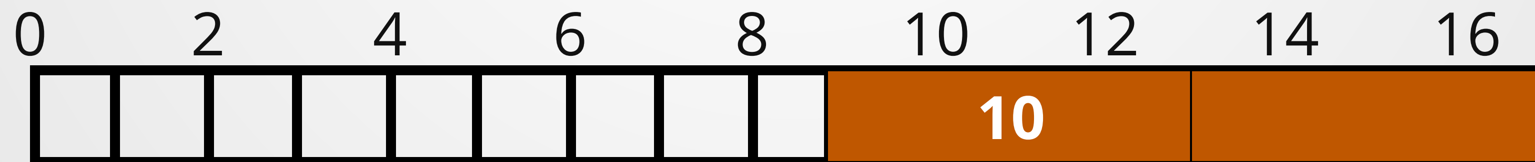
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9

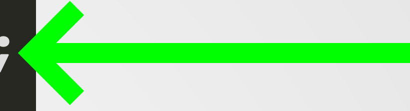


```
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

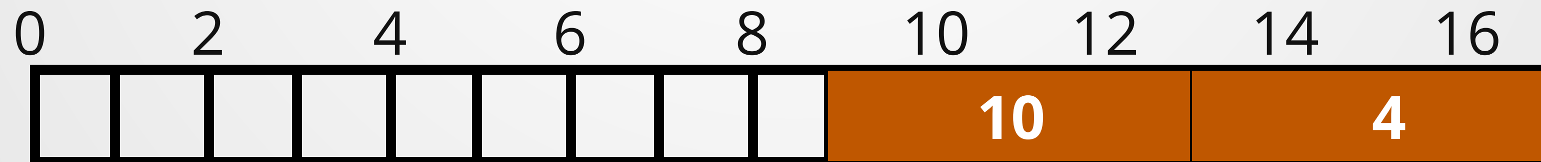
Name	Type	Address
x	int	9



```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

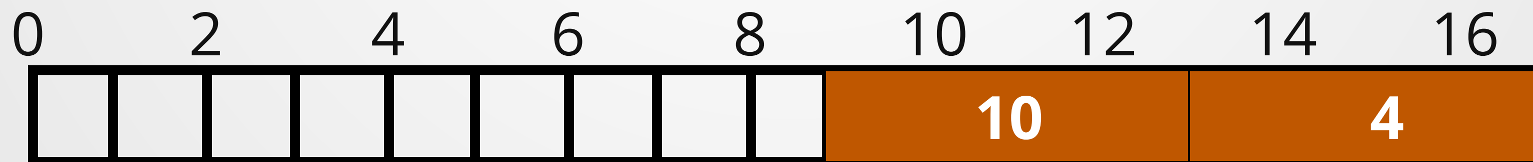


Name	Type	Address
x	int	9



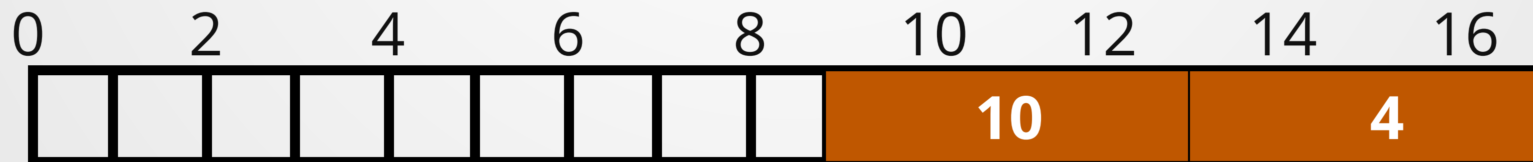
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9
sz_x	size_t	13



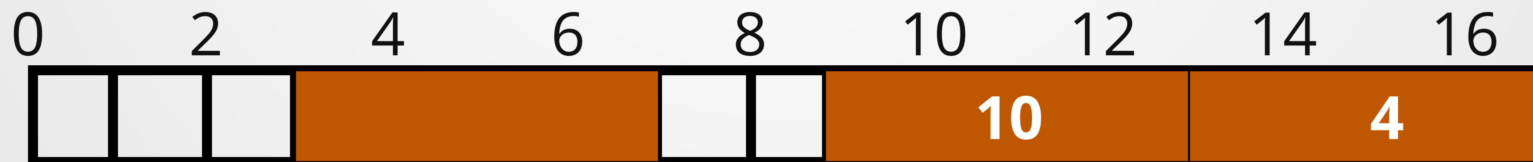
```
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9
sz_x	size_t	13



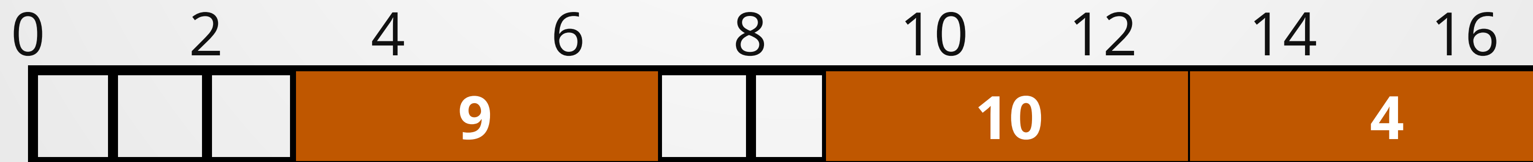
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9
sz_x	size_t	13



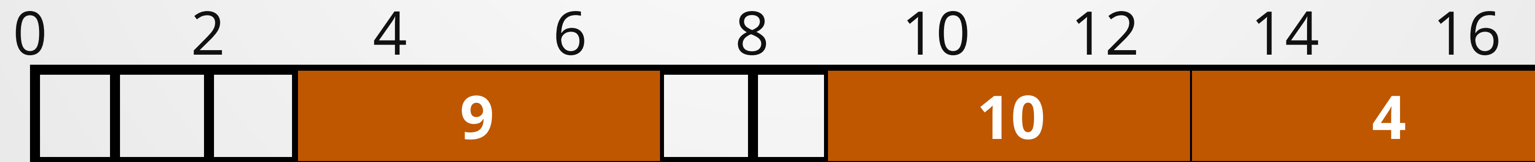
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9
sz_x	size_t	13



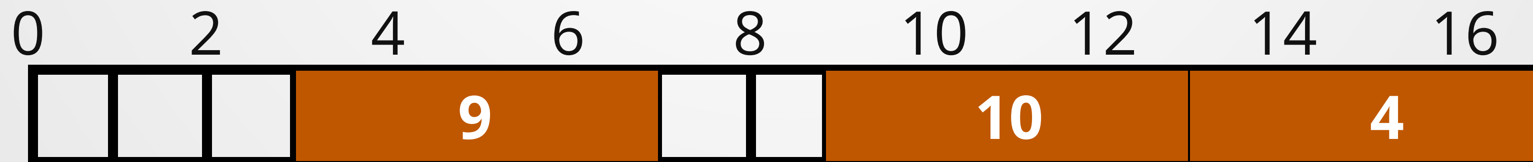
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
```

Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3



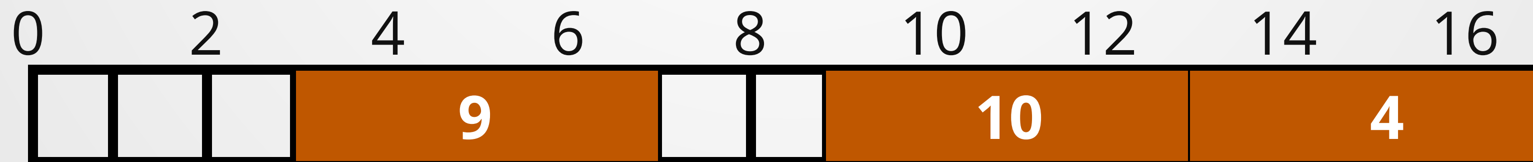
```
1 int x = 10;
2 size_t sz_x = sizeof(x);
3 int* addr_x = &x;
4 sz_x = 3;
```

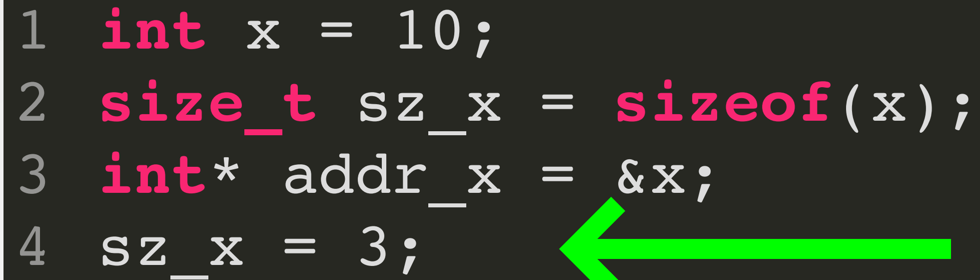
Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3



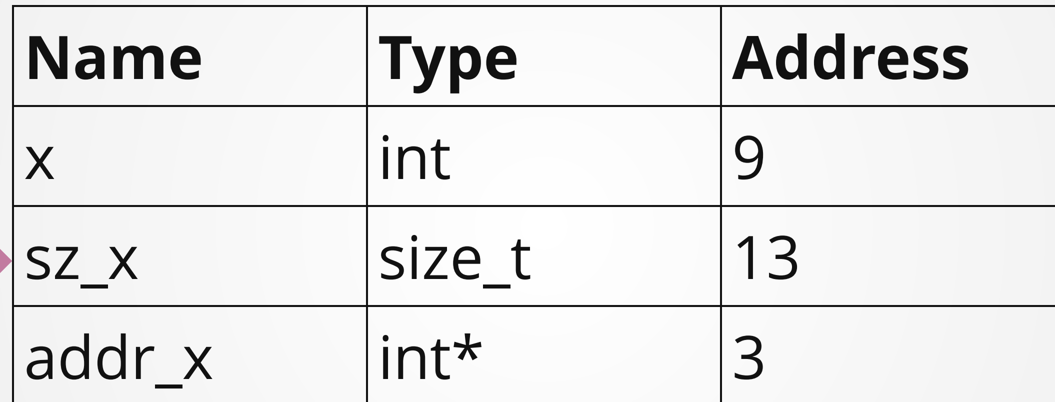
```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
4  sz_x = 3;
```

Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3

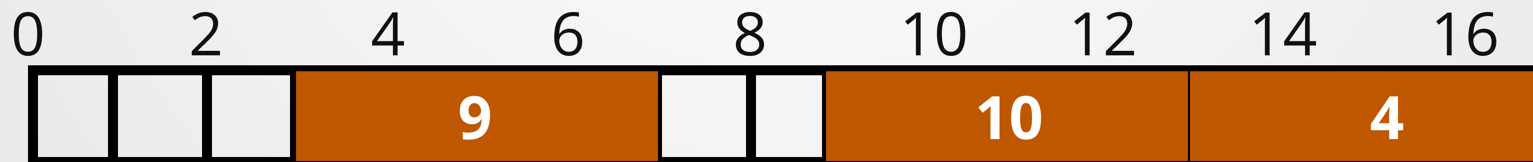




```
1 int x = 10;
2 size_t sz_x = sizeof(x);
3 int* addr_x = &x;
4 sz_x = 3;
```

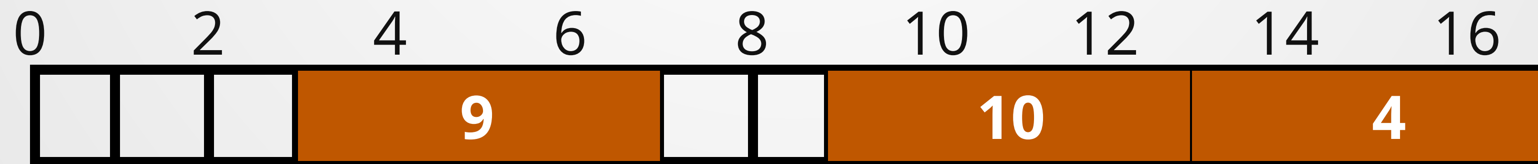


Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3



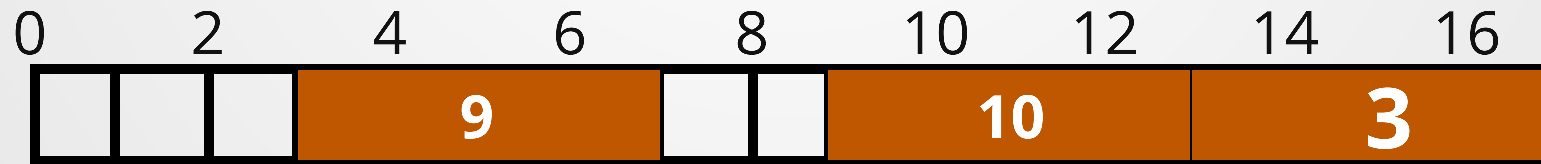
```c
int x = 10;
size_t sz_x = sizeof(x);
int* addr_x = &x;
sz_x = 3;
```

Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3



```c
1  int x = 10;
2  size_t sz_x = sizeof(x);
3  int* addr_x = &x;
4  sz_x = 3;
```

Name	Type	Address
x	int	9
sz_x	size_t	13
addr_x	int*	3



Pointers 101

Pointer Types

```
1 int* x;
2 float* y;
3 char* z;
4 Dog* fido;
```

Any type X can be modified by adding a *, denoting a pointer to that type.

Pronounced "pointer-to-X", or if the type is not important, just "pointer".

```
1 // This is a pointer-to-Dog and a Dog
2 Dog* old_dan, little_ann;
3
4 // This is two pointer-to-Dogs
5 Dog *old_dan, *little_ann;
```

Operations on Pointers

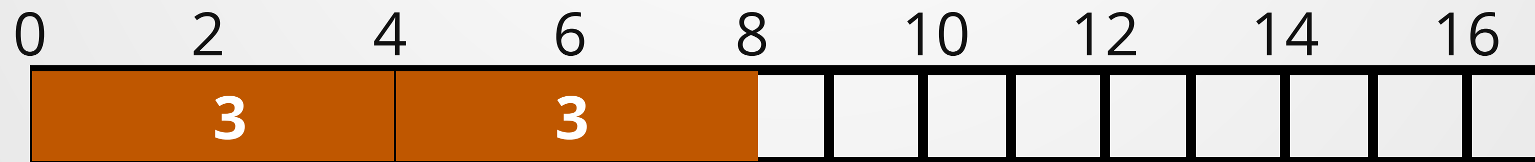
Dereference: read the address, then read/write that address.

```
1 int x = 3;
2 int y = x;
3 y = 10;
4 std::cout << x;
```

```
1 int x = 3;
2 int* p = &x;
3 *p = 10;
4 std::cout << x;
```

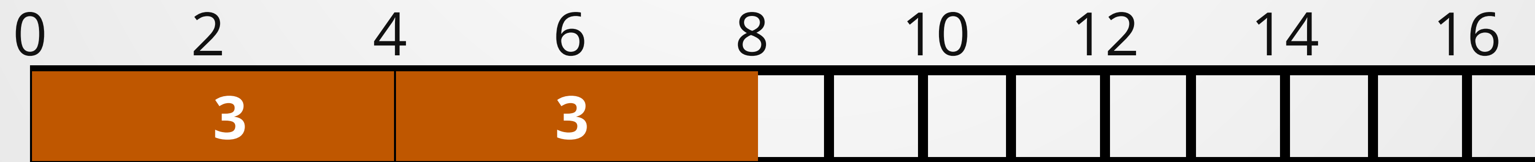
```cpp
1 int x = 3;
2 int y = x;
3 y = 10;
4 std::cout << x;
```

Name	Type	Address
x	int	0
y	int	4



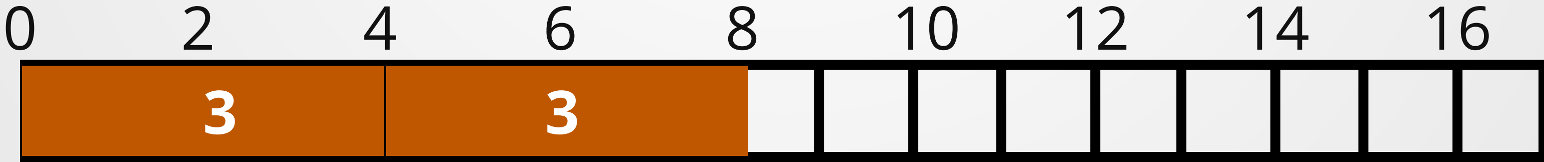
```cpp
int x = 3;
int y = x;
y = 10;
std::cout << x;
```

Name	Type	Address
x	int	0
y	int	4



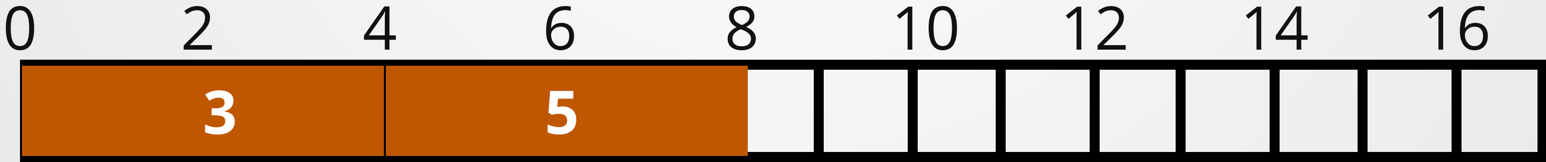
```cpp
1  int x = 3;
2  int y = x;
3  y = 10;
4  std::cout << x;
```

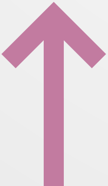
Name	Type	Address
x	int	0
y	int	4



```cpp
1 int x = 3;
2 int y = x;
3 y = 10;
4 std::cout << x;
```

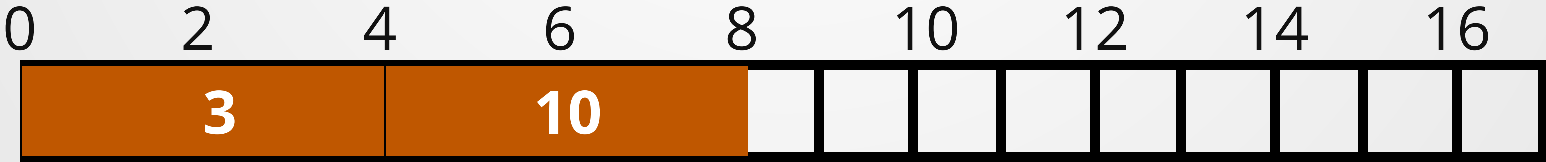
Name	Type	Address
x	int	0
y	int	4

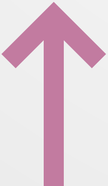


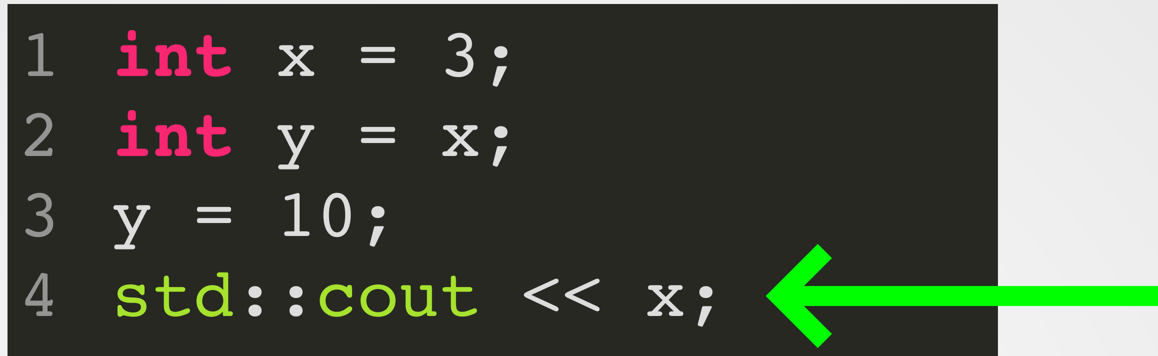


```
1 int x = 3;
2 int y = x;
3 y = 10;
4 std::cout << x;
```

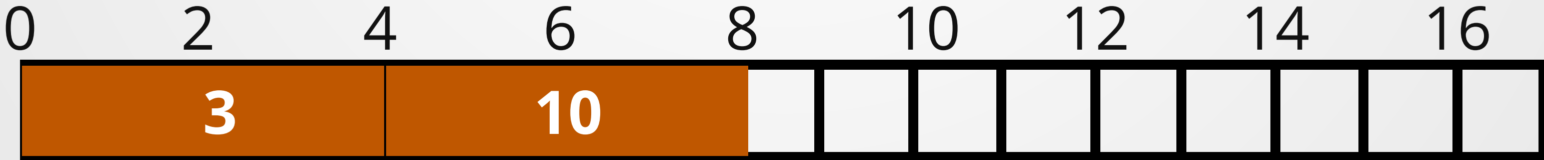
Name	Type	Address
x	int	0
y	int	4





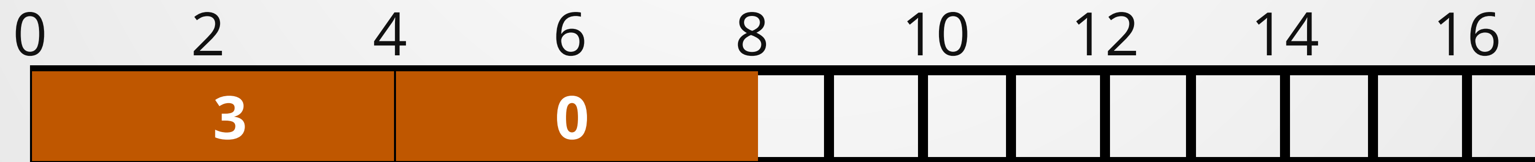


Name	Type	Address
x	int	0
y	int	4



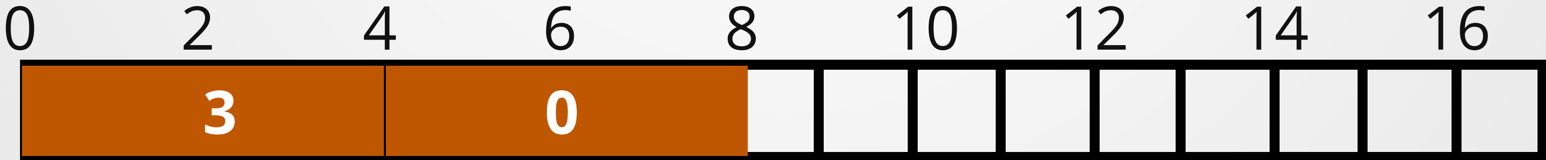
```cpp
int x = 3;
int* p = &x;
*p = 10;
std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



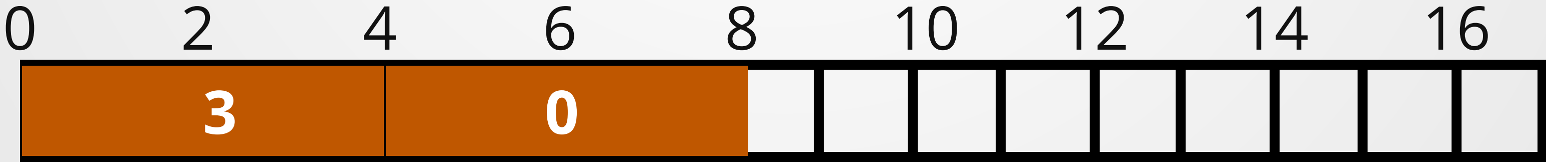
```cpp
int x = 3;
int* p = &x;
*p = 10;
std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



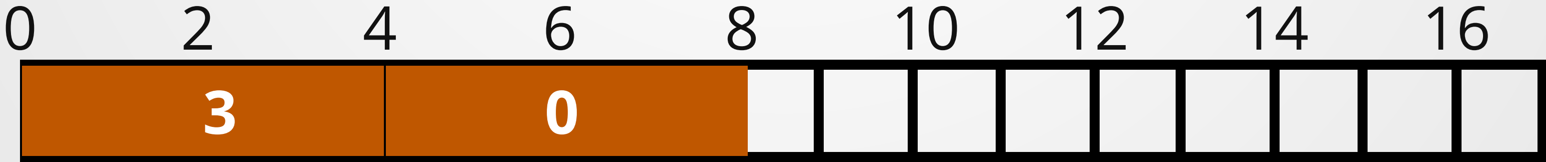
```cpp
1  int x = 3;
2  int* p = &x;
3  *p = 10;
4  std::cout << x;
```

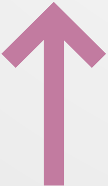
Name	Type	Address
x	int	0
p	int *	4



```
1 int x = 3;
2 int* p = &x;
3 *p = 10;
4 std::cout << x;
```

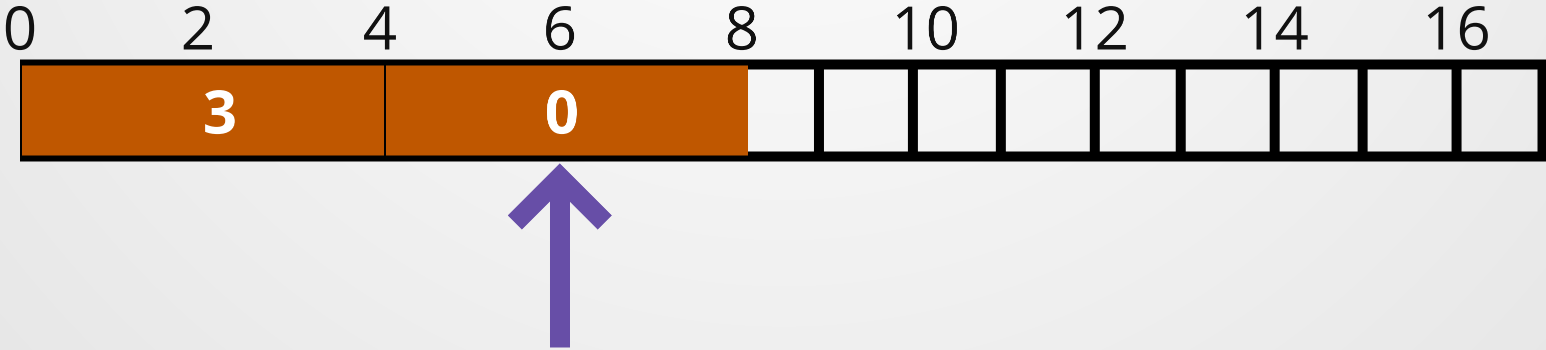
Name	Type	Address
x	int	0
p	int *	4





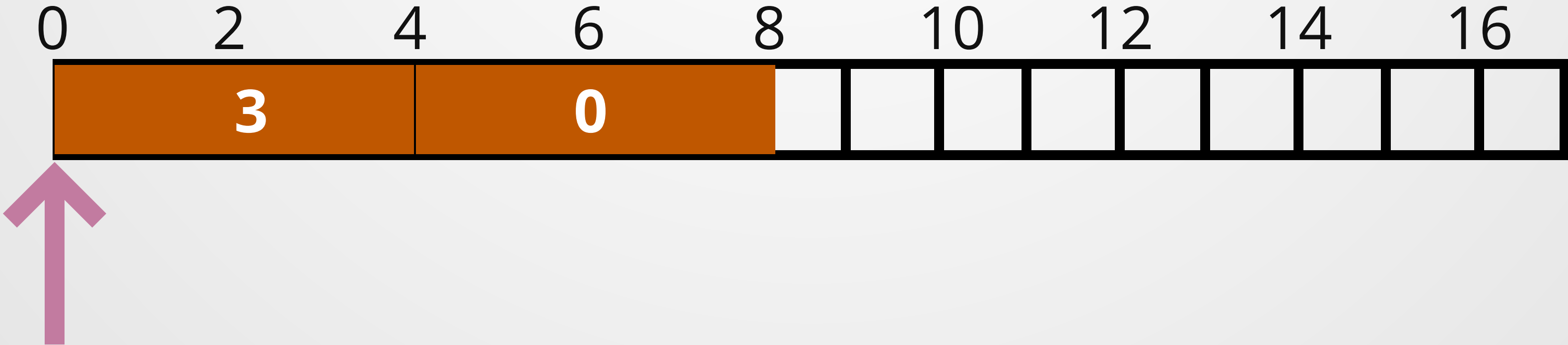
```
1  int x = 3;
2  int* p = &x;
3  *p = 10;
4  std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



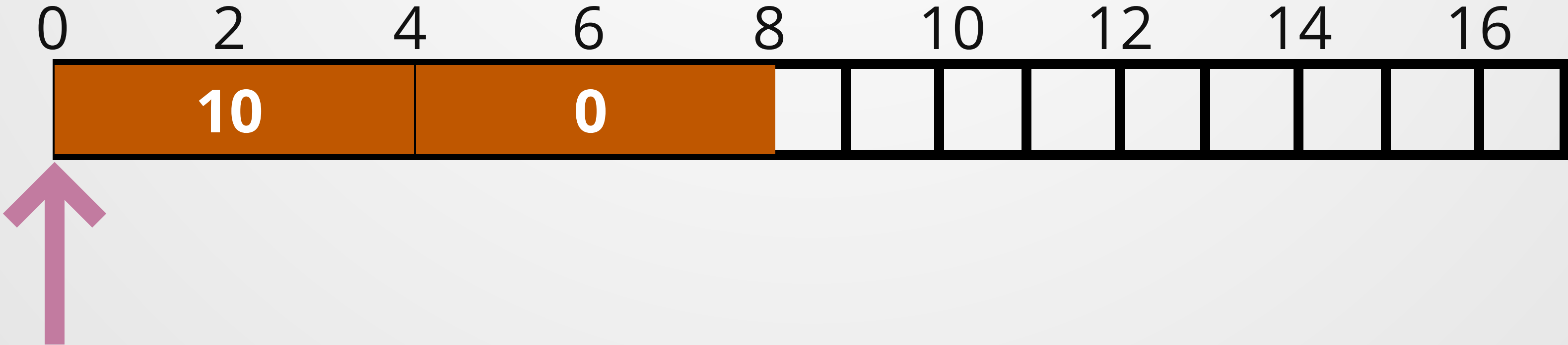
```cpp
int x = 3;
int* p = &x;
*p = 10;
std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



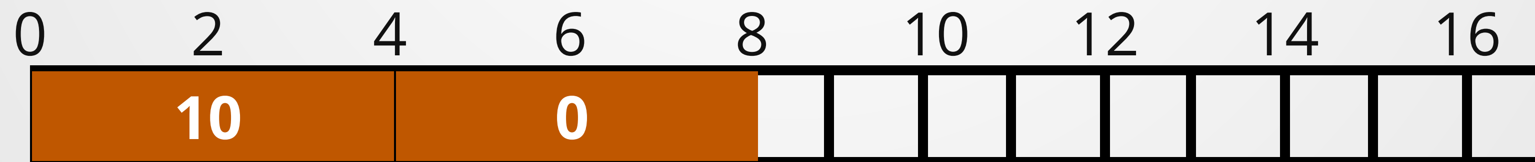
```cpp
1 int x = 3;
2 int* p = &x;
3 *p = 10;
4 std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



```cpp
1  int x = 3;
2  int* p = &x;
3  *p = 10;
4  std::cout << x;
```

Name	Type	Address
x	int	0
p	int *	4



Pass-by-pointer

```
1 void changeInput(int x){
2   x = x + 5;
3 }
4
5 // Doesn't work
```

```
1 void changeInput(int* x){
2   *x = *x + 5;
3 }
4
5 // Changes its input
```

A Very Special Pointer

A *null pointer* is a pointer which does not point to anything.

Dereferencing a null pointer is undefined behavior.

```
1  int* null1 = NULL; // Do this in older codebases
2  int* null2 = nullptr; // Do this for C++11 and on
```

Problem!

What state is this code in after each line 7,8,9,10? Assume variables are 4 bytes large and allocated in the declared order.

```
1  int** ptrptr = nullptr;
2  int* ptr1 = nullptr;
3  int* ptr2 = nullptr;
4  int x = 3;
5  int y = 4;
6
7  ptr1 = &y;
8  ptr2 = &x;
9  ptrptr = &ptr2;
10 **ptrptr = 2;
```

Pointers and UB

It is **very** easy to accidentally invoke UB using pointers.

Things that can invoke this include:

- Dereferencing a null pointer
- Dereferencing an invalid pointer
- Accessing memory through an incorrectly-typecasted pointer
- Using pointers to modify constant variables
- Comparing pointers in the "wrong" manner
- Using a pointer after the memory it points to is no longer valid

...and many more!

```c
1 int main(){
2       int *x;
3       *x = 2000;
4       // what did I just write to??
5 }
```

```
Instructor@CS105C
>
```

Pointers

There is a *lot* more to learn about pointers, but this will do for now.

- Declared with type*
- A number that refers to (points to) a byte in memory
- Get pointer to value by using address-of (&) operator
- Get value pointed-to by using dereference (*) operator
- Can be dangerous: NULL pointers or bad pointers cause awful behavior

References

References

A large number of pointer bugs arise because of two facts about pointers:

- Pointers can be NULL
- Pointers can be arbitrarily reassigned

But they're still very powerful! Can we create a construct that offers 99% of the power of pointers while being much, much safer to use?

References

```cpp
int x = 3;
int& r = x;
r = 10;
std::cout << x;

// Prints 10
```

```cpp
int x = 3;
int* p = &x;
*p = 10;
std::cout << x;

// Prints 10
```

ref_x is another name for x

References

```
1 int x = 3;
2 int& ref_x = x;
3 ref_x = 10;
4 std::cout << x;
```

Name	Type	Address
x	int	0
r	int &	0

Pass-by-Reference

```cpp
void changeInput(int x){
  x = x + 5;
}

// Doesn't work
```

```cpp
void changeInput(int& x){
  x = x + 5;
}

// Changes its input
```

Reference vs. Pointer

References	Pointers
• Must be initialized on creation • Cannot be NULL (or UB happens) • Cannot be changed once bound	• Can be uninitialized • Can be NULL • Can be reassigned at will

C++ Classes + Objects

Part 1

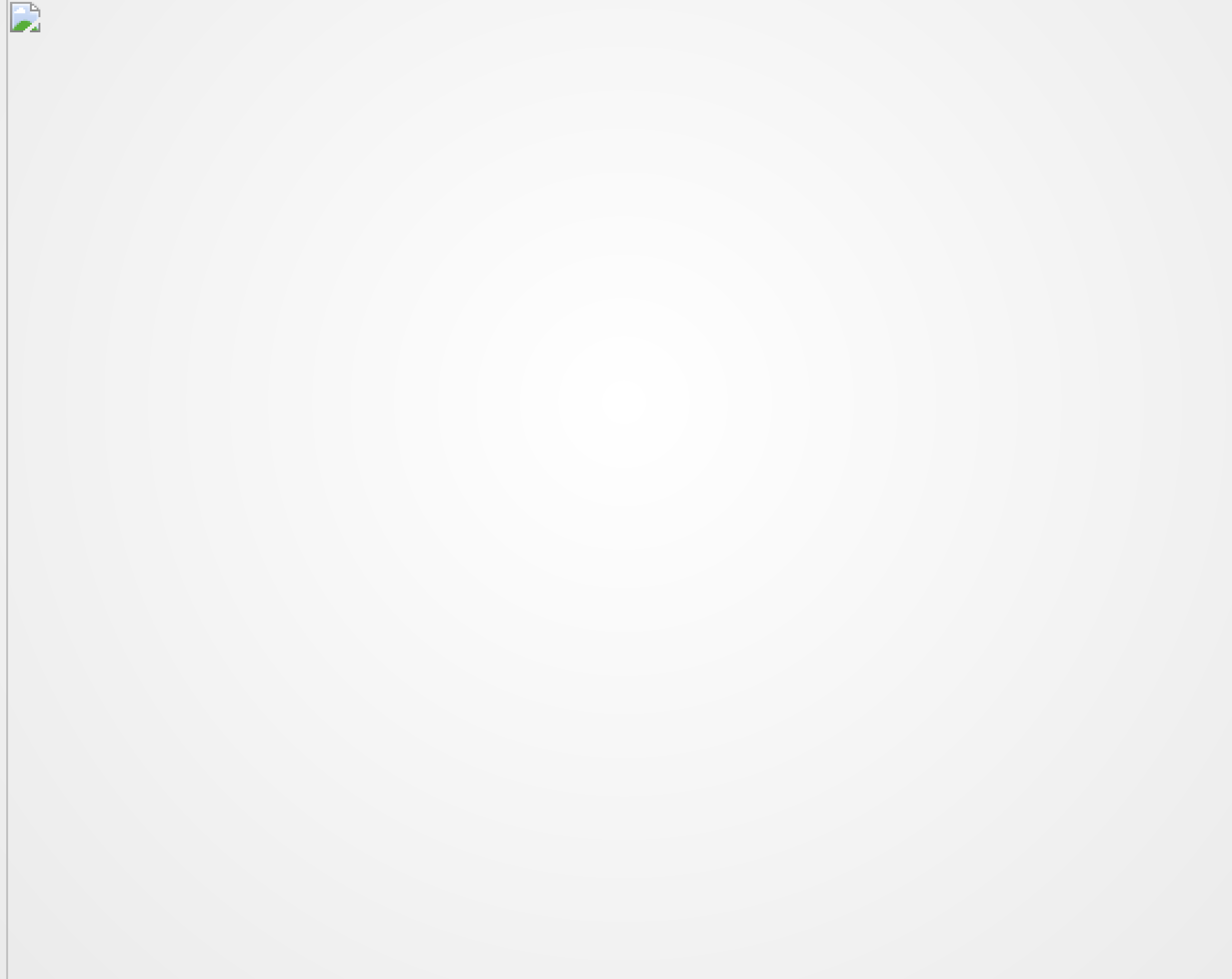
What are classes good for?

- Encapsulation: group related functions/data together
- Abstraction: allow user of code to specify *what* should be done instead of *how* to do it.

Example Class Declaration

```cpp
class Class {
  public:
    int x;
    float y;

    Class();
    Class(int);
    void somethingPublic();

  private:
    int* other;
    int& other2;
    void somethingPrivate();
};  // Don't forget semicolon!
```

Example: Space Invaders



Attempt #1

```
1  int main(){
2    std::vector<double> laser_x;
3    std::vector<double> laser_y;
4
5    std::vector<double> alien_x;
6    std::vector<double> alien_y;
7
8    // Populate aliens
9    for(int i = 0; i < MAX_ALIENS; i++){
10     alien_x.push_back( gen_initial_alien_x(i) );
11     alien_y.push_back( gen_initial_alien_y(i) );
12   }
13
14   while(true){
15     readInput();
16     update(laser_x,laser_y,alien_x,alien_y);
17   }
18 }
```

Issues:

- Possible to desynchronize the lists
- Maybe coordinates need to be checked before writing!
- Conceptually, not grouping related data together.
- No simple way to initialize a new laser/alien.

```cpp
struct Point{
  float xpos;
  float ypos;
};

class Laser {
  public:
    Point pos;
    float length;
};

class Alien {
  public:
    Point pos;
    float size;
};

int main(){
  std::vector<Alien> aliens;

  // Create all aliens at the start of the game
  for(int i = 0; i < ALIEN_MAX; i++){
    Alien a;
    Point p;
    p.xpos = 3.0;
    p.ypos = 4.0;
    a.pos = p;
    a.length = 1.0;
    aliens.push_back(a);
  }
}
```

Solved:

- Possible to desynchronize the lists
- Not grouping related data together

Not solved:

- No simple way to initialize a new laser/alien.

Constructors

Special functions whose job it is to construct the object.
These can be overloaded like normal functions.

```
1 class Point{
2   float xpos;
3   float ypos;
4
5   Point();
6   Point(float x, float y);
7   Point(const Point& other);
8 };
```

How to use a Constructor

Two ways to call a constructor:

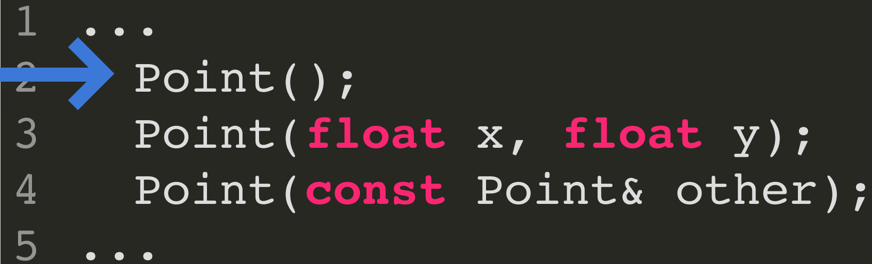
```
Point p1(1.0,2.0);

Point p2 = Point(1.0, 2.0);
```

Do not do the following!

```
// Doesn't compile
Point point = new Point(1.0, 2.0);
```

The 'new' keyword in C++ has a very special meaning--do not just randomly sprinkle it in, or things will go bad very quickly!

Special Constructors



Default Constructor

If the class is ever initialized with a default value, the default constructor is called.

```
1 // Creates a vector with 100 lasers
2 std::vector<Point> Points(100);
3
4 // Each element is initialized by calling
5 // the default constructor.
```

Special Constructors

```
...
  Point();
  Point(float x, float y);
  Point(const Point& other);
...
```

Copy Constructor

Responsible for implementing a value copy.

```
Point p1(1.0,1.0);

Point p2 = p1;

// Thanks to one-argument constructor rule,
// the above is equivalent to
Point p2(p1);
```

Example Implementations

```
1 Point::Point() : xpos(0), ypos(0) { }
2
3 Point::Point(float x, float y) : xpos(x), ypos(y) { }
4
5 Point::Point(const& Point o) : xpos(o.xpos), ypos(o.ypos) { }
```

...or maybe we're feeling lazy.

```
1 class Point{
2   float xpos;
3   float ypos;
4 
5   Point() = default;
6   Point(float x, float y); // Still need to implement this
7   Point(const Point& other) = default;
8 };
```

```cpp
class Point{
  float xpos;
  float ypos;

  Point() = default;
  Point(float x, float y);
  Point(const Point& other) = default;
};

class Laser { ... };

class Alien {
    ...
    Alien() = default;
    Alien(Point p);
};

int main(){
  Point p1(1.0,2.0);
  Alien sane(p1);

  // OR!
  Alien sane(Point(1.0,2.0));
}
```

```cpp
class Point{
  float xpos;
  float ypos;

  Point() = default;
  Point(float x, float y);
  Point(const Point& other) = default;
};

class Laser { ... };

class Alien {
    ...
    Alien() = default;
    Alien(Point p);
};

int main(){
  Alien crazy(Point(1000000, -1110010101));
}
```

Solved:

- Possible to desynchronize the lists
- Not grouping related data together
- Simple way to initialize a new laser/alien.

Not Solved:

- Position checks!

Methods

The Problem

Our game entities have to remain within some fixed box: the positions can't be arbitrary!

Secondary, but still important: need to resolve collisions (laser collision with alien = remove alien)

Solution: Write a method to check!

```
1  struct Point{
2    float xpos;
3    float ypos;
4
5    bool isInbounds(Point lowerleft,
6                    Point upperright);
7  };
8
9  ...
10
11 bool Point::isInbounds(Point ll, Point ur){
12   if( xpos < ll.xpos ){ return false; }
13   if( ypos < ll.ypos ){ return false; }
14   ...//etc. etc.
15   if( ypos > ur.ypos ){ return false; }
16   return true;
17 }
```

```cpp
class Laser {
  public:
    Point pos;
    float length;
    Laser() = default;
    Laser(Point p);
    Laser(const Laser& other) = default;
};

class Alien {
  public:
    Point pos;
    float size;
    Alien() = default;
    Alien(Point p);
    Alien(const Alien& other);
};
```

As long as these values are public, we can **never** assume that the position is valid!

Why?

```cpp
Alien a(Point(1.0,2.0));
a.pos = Point(100000,300000);
a.run_game_logic();
```

```cpp
class Laser {
    Point pos;
    float length;

  public:
    Laser() = default;
    Laser(float x, float y, float len=1.0);
    Laser(const Laser& other) = default;

    Point getPosition();
    void  setPosition(); // Careful about error design
};

class Alien {
    Point pos;
    float size;

  public:
    Alien() = default;
    Alien(float x, float y);
    Alien(const Alien& other);

    Point getPosition();
    void  setPosition(); // Careful about error design
};
```

```cpp
Alien a(Point(1.0,2.0));
a.pos = Point(100000,300000);
a.run_game_logic();
```

```
❯ g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:22:5: error: ‘Point Alien ::pos’ is private within this context
   22 |   a.pos = Point(100000,300000);
      |     ^ ~~
test.cpp:8:11: note: declared private here
    8 |     Point pos;
      |           ^ ~~
```

Operators

```cpp
class World {
  ...
  Point alienBlock;
  std::vector<Alien> aliens;
}

class Alien {
  Point offset;
  Point* block;

  ...
  Point getPosition(){
    // Somehow need to add the offset and the block.
  }
}
```

Aliens move as a block.

Maybe it's easier to have a single Point represent the position of the block, then have each alien's position represent an offset from that block.

Solution with Methods

```
1  class Point{
2    ...
3    Point add(Point other);
4    ...
5  };
6
7  Point Point::add(Point other){
8    float x = xpos + other.xpos;
9    float y = ypos + other.ypos;
10   return Point(x,y);
11 }
12
13 // and a similar method for scale
```

```
1  int main(){
2    Point p1, p2, p3, p4;
3    // Initialize our points
4
5    Point pfinal = p1.add(p2.scale(2).add(p3)).add(p4);
6  }
```

```
1  int main(){
2    Point p1, p2, p3, p4;
3    // Initialize our points
4
5    Point pfinal = p1 + (p2 * 2 + p3) + p4;
6  }
```

Let's use operators instead!

```cpp
class Point{
  ...
  Point add(Point other);
  ...
};

Point Point::add(Point other){
  float x = xpos + other.xpos;
  float y = ypos + other.ypos;
  return Point(x,y);
}

// and a similar method for scale
```

```cpp
class Point{
  ...
  Point operator+(Point other);
  ...
};

Point Point::operator+(Point other){
  float x = xpos + other.xpos;
  float y = ypos + other.ypos;
  return Point(x,y);
}

// and a similar method for scale
```

```cpp
int main(){
  Point p1, p2;
  // Initialize points

  Point p3 = p1 + p2;
  Point p4 = p1.add(p2);
}
```

This technique of specializing behavior of an operator is known as **operator overloading**, because you're overloading the behavior of the operator based on types.

The first argument to the operator is the class whose operator method will be invoked. The second is the argument(s) to the operator.

```
1  struct Point{ ... };
2
3  std::ostream &operator<<(std::ostream &os, Point p) {
4    os << "(";
5    os << p.xpos;
6    os << ", ";
7    os << p.ypos;
8    os << ")";
9    return os;
10 }
11
12 int main(){
13   Point p1(2.0,3.0);
14   Point p2(0.0,-3.0);
15   std::cout << p1 << '\n'
16             << p2 << std::endl;
17 }
```

```
❯ ./a.out
(2, 3)
(0, -3)
```

Still a few strange warts...

```
1 int main(){
2   Point p1, p2, p3, p4;
3   // Initialize our points
4
5   Point pfinal = p1 + (p2 * 2 + p3) + p4;
6 }
```

```
1 int main(){
2   Point p1, p2, p3, p4;
3   // Initialize our points
4
5   Point pfinal = p1 + (2 * p2 + p3) + p4;
6 }
```

```
1 int main(){
2   Point p1, p2, p3, p4;
3   // Initialize our points
4
5   Point pfinal = p1.add(p2.scale(2).add(p3)).add(p4);
6 }
```

Inheritance

Inheritance is a mechanism to reuse code and model problems.

```
1 class Animal { ... };
2
3 class Dog : public Animal { ... };
4
5 class Cat : public Animal { ... };
```

Overriding

Subclasses can *override* the methods of their superclass.

```cpp
class Animal {
    ...
    void makeNoise(){
      std::cout << "I'm an animal" << '\n';
    }
}

class Dog : public Animal {
    ...
    void makeNoise(){
      std::cout << "I'm a Dog" << '\n';
    }
}

int main(){
  Animal a;
  Dog    b;

  a.makeNoise();
  b.makeNoise();
}
```

...but something's funny here.

```
1  class Animal {
2      ...
3      void makeNoise(){
4        std::cout << "I'm an animal" << '\n';
5      }
6  }
7
8  class Dog : public Animal {
9      ...
10     void makeNoise(){
11       std::cout << "I'm a Dog" << '\n';
12     }
13 }
14
15 int main(){
16   Animal a;
17   Dog    b;
18   Animal c = Dog();
19
20   a.makeNoise();
21   b.makeNoise();
22   c.makeNoise();
23 }
```

```
user@CS105C>
```

Summary

Pointers

A method for manipulating memory in C++

Dereferencing the pointer tells the language to use the memory location "pointed to" by the pointer.

Have their own special syntax and types.

Very easy to accidentally invoke UB with.

References

A modern alternative to pointers in C++

Allow us to have many of the benefits of pointers without their drawbacks.

Have a special type, but once declared, can be used like normal variables.

Classes

A mechanism to bundle related data/functions together while hiding some of it from outside eyes.

Constructors

A special function whose responsibility is to initialize an object.

Some special constructors are used by the language in some situations.

We can (should) use special initialization syntax in the constructor.

Methods

A function attached to a class. It can access all the classes private members.

Can be written as an **operator overload**, in which case it will be called if the appropriate operator is called on the class.

Announcements!

Project 0 is due next week

Notecard:

- EID and Name
- Something new you learned (can be "nothing")
- A question you have (**<u>cannot</u>** be "none")