EECS 280 – Lecture 5

Compound Objects

1/24/2022

Agenda

- `const`
  - `const` in declarations
  - `const` conversions
- Compound (Class-Type) Objects
  - Basics
  - Members
  - Passing compound parameters.

Oops.

- What's wrong with this code?

```
void strcpy(char *dst, const char *src) {
  while (*src != '\0') {
    *src = *dst;
    ++src;
    ++dst;
  }
  *src = *dst;
}

int main() {
  char str1[6] = "hello";
  char str2[6] = "apple";
  strcpy(str1, str2); // str1 array now holds "apple"
}
```

This const means the compiler will reject the faulty assignments.

The assignments are backwards.

The const Keyword

- We tell the complier we never intend to modify something, and it keeps us honest.

```
void strcpy(char *dst, const char *src);
```

- const is a **type qualifier**.
- const forbids **assignment**.
  - Initialization: OK (first value)
  - Assignment: NOT ALLOWED!

```
const int x = 3; // ok
x = 5; // won't compile
```

const in Declarations

- As always, read from the inside out.

These are equivalent!

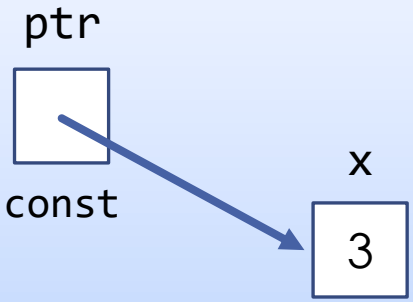
```
int const * arr[6];
const int * arr[6];
```

"arr is an array of 6 pointers to const ints"

const pointer vs. pointer-to-const

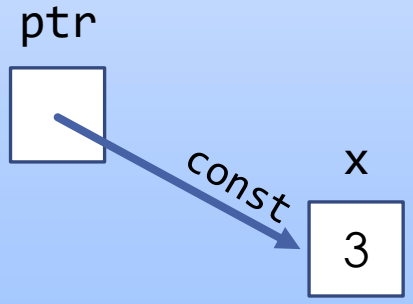
- const pointer
  - The pointer value (an address) itself cannot change.

```
int x = 3;
int * const ptr = &x;
```



- Pointer-to-const
  - You can't use the pointer to change the object.

```
int x = 3;
int const * ptr = &x;
const int * ptr = &x;
```

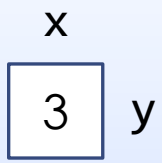


Note: You can also have both: a const pointer-to-const.

reference-to-const

- A **reference** creates an **alias** for an object.

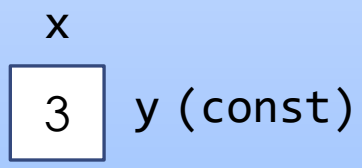
```
int x = 3;
int &y = x;
y = 10; // legal, changes x too
```



- **Reference-to-const**

  The alias can't be used to change the object.

```
int x = 3;
int const &y = x;
y = 10; // error
x = 10; // still legal
```



References can never be re-bound, so a literal const reference isn't meaningful. However, "const reference" is often used loosely to mean "reference-to-const".

http://bit.ly/3a5lsz7

Exercise: const

- Which of the assignments are legal?

```
int x = 3;
int y = 4;
int const * a = &x;
int const b = x;
int * const c = &x;
int const &d = x;

*a = 5;
b = 5;
*c = 5;
c = &y;
d = y;
a = &b;
x = 5;
```

Question

How many of these assignments are legal (will not cause a compiler error)?

A) 2

B) 3

C) 4

D) 5

E) 6

Exercise: const

- Which of the assignments are legal?

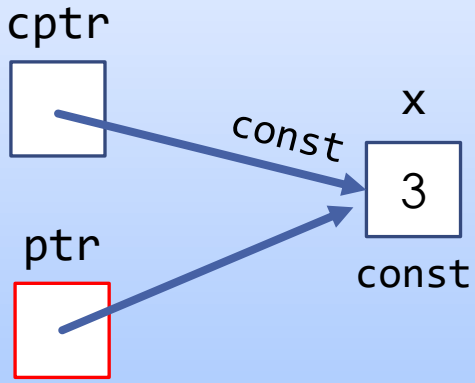
```
int x = 3;
int const * a = &x;
int const b = x;
int * const c = &x;
int const &d = x;

*a = 5;
b = 5;
*c = 5;
c = &x;
d = x;
a = &b;
x = 5;
```

const conversions

```
int const x = 3;
int const *cptr = &x;
int *ptr = cptr; // this line
```

- Are all the const objects still "safe"?
- Answer: No.

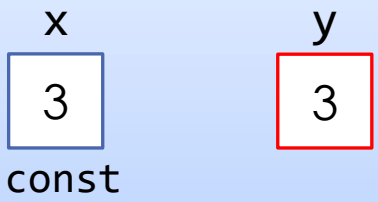


- We can NOT…
  - …convert a pointer-to-const to a regular pointer.

const conversions

```
int const x = 3;
int y = x; // this line
```

- Are all the const objects still "safe"?
- Answer: Yes.

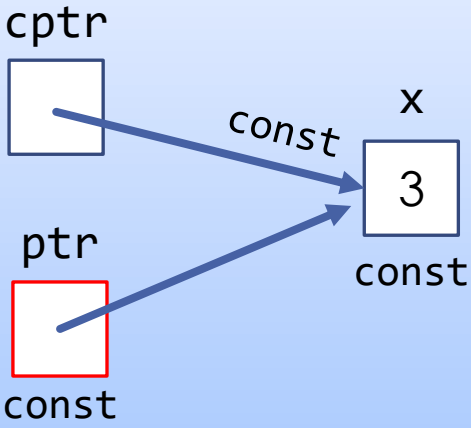


- We can…
  - …convert a const value into a regular value. (We're just making a copy.)

const conversions

```
int const x = 3;
int const *cptr = &x;
int * const ptr = cptr; // this line
```

- Are all the const objects still "safe"?
- Answer: No.



- We can NOT…
  - …convert a pointer-to-const to a const pointer.

http://bit.ly/3aMA7hr

Exercise: const conversions

- Which of the assignments are legal?

```
int x = 3;
const int y = 5;
int *ptr = &x;
int const *cptr = &y;
int &ref = x;
int const &cref = y;

x = y;
ptr = &y;
cptr = ptr;
cref = &x;
cref = *ptr;
x = *cptr;
```

Question

How many of these assignments are legal (will not cause a compiler error)?

A) 2

B) 3

C) 4

D) 5

E) 6

Exercise: const conversions

- Which of the assignments are legal?

```
int x = 3;
const int y = 5;
int *ptr = &x;
int const *cptr = &y;
int &ref = x;
int const &cref = y;

x = y;
ptr = &y;
cptr = ptr;
cref = &x;
cref = *ptr;
x = *cptr;
```

const and Functions

http://bit.ly/3aMA7hr

- For any function call, the compiler also has to make sure to protect **const** objects.

```
void strFunc1(const char *str);
void strFunc2(char *str);
void intFunc3(int a);
```

```
int main() {
  const char strA[6] = "hello";
  char strB[6] = "apple";
  const int num = 3;

  strFunc1(strA);
  strFunc1(strB);

  strFunc2(strA);
  strFunc2(strB);

  intFunc3(num);
}
```

Question

Which of the function calls in `main()` is "sus"?

*When the compiler notices a "sus" call, it gives an error.

Agenda

- `const`
  - `const` in declarations
  - `const` conversions
- **Compound (Class-Type) Objects**
  - Basics
  - Members
  - Passing compound parameters.

Kinds of Objects in C++

- **Atomic**
  - Also known as **primitive**.
  - `int`, `double`, `char`, etc.
  - Pointer types.
- **Arrays** (homogeneous)
  - A *contiguous* sequence of objects of the same type.
- **Class-type** (heterogeneous)
  - A compound object made up of **member** subobjects.
  - The members and their types are defined by a **struct** or **class**.

Compound Objects

- We can use both `struct` and `class` to create class-type objects in C++.
- We'll focus on `struct` for now.

The struct definition creates a new type called `Person`.

In `main`, we create some local `Person` objects, but they're not initialized.

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  int x;
  Person alex;
  Person jon;
}
```

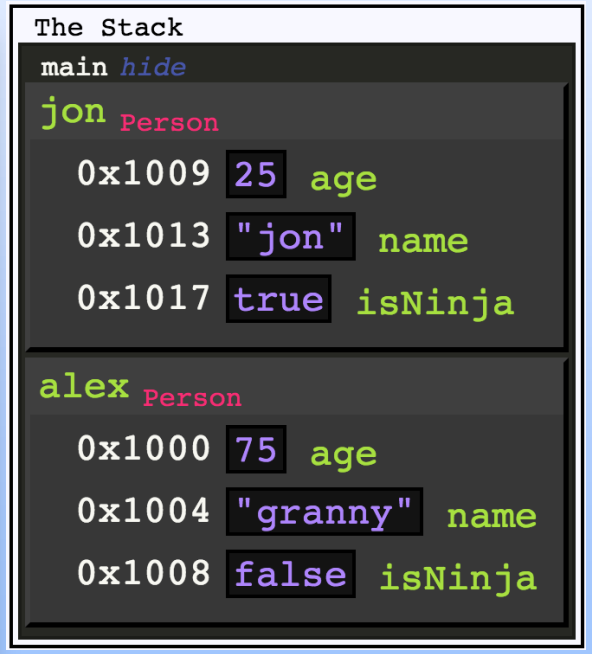
Member declarations define the subobjects a compound object has.

Initializing `structs`

- You can use an initializer list to initialize each member of a `struct`.
  - You can also do this for assignment, unlike with an array.

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person alex;
  Person jon = { 25, "jon", true };
  alex = { 75, "granny", false };
}
```

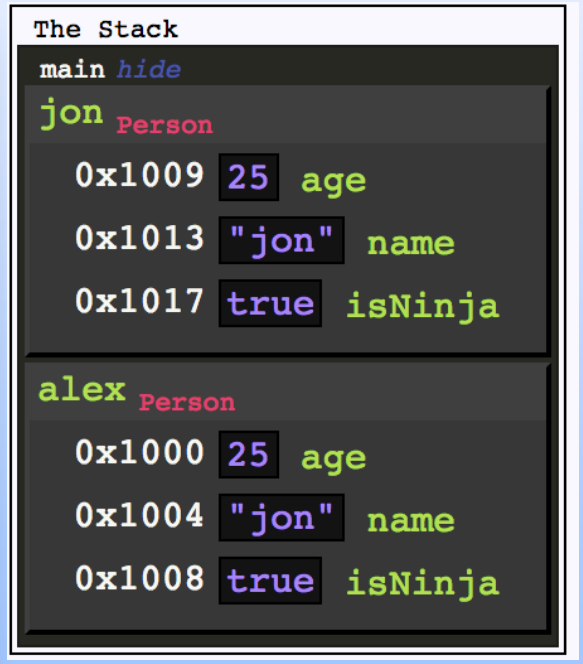


structs and Value Semantics

- Unlike arrays, structs do have a "value".[1]
- Copying a struct value just copies each member one by one.[2]

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person alex;
  Person jon = { 25, "jon", true };
  alex = jon;
}
```



1 Hurray! No more turning into pointers!
2 We'll see more details when we discuss copy constructors.

structs and const

- A struct can be declared **const**. Neither it nor its members may be assigned to.

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  const Person p1 = { 17, "Kim", true };
  Person p2 = { 17, "Ron", true };

  p1.isNinja = false; // not possible
  p1 = p2; // not possible
}
```

structs and pointers

- A pointer can point to a struct.
- Think of this as pointing to the "whole" struct.
- Dereference, then use the . to get members
  - Alternatively, use the -> operator as a shortcut!

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person p = { 31, "Aliyah", true };
  Person * ptr = &p;
  p.age = 32;
  (*ptr).age = 33;
  ptr->age = 34;
}
```

obj . member

ptr -> member

Break Time!

These are examples of resized images students and staff came up with in previous terms (using project 2 resize).

Break Time!

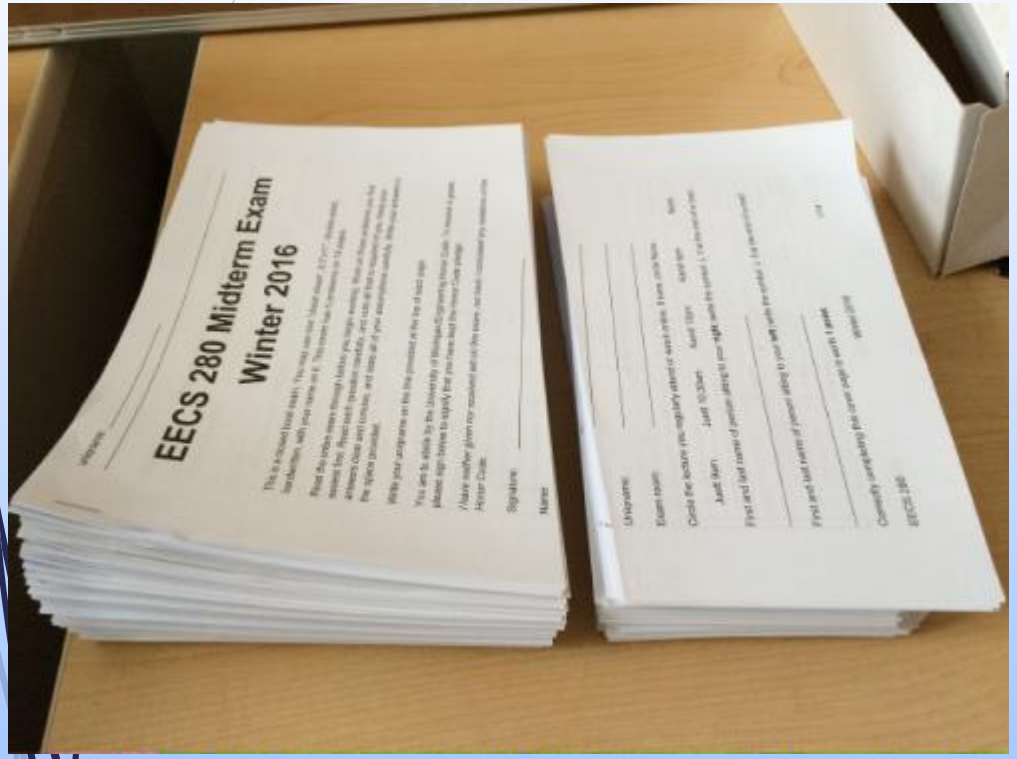
These are examples of resized images students and staff came up with in previous terms (using project 2 resize).

Break Time!

These are examples of resized images students and staff came up with in previous terms (using project 2 resize).

Break Time!

These are examples of resized images students and staff came up with in previous terms (using project 2 resize).



Break Time!

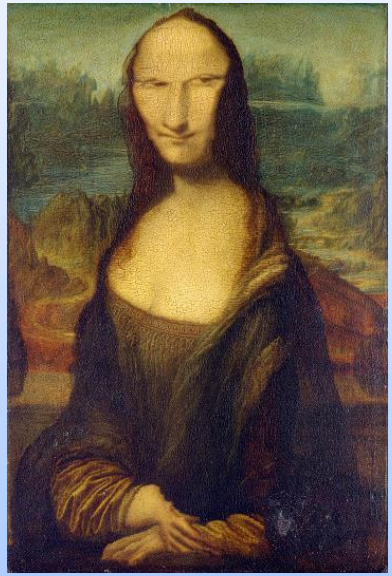
These are examples of resized images students and staff came up with in previous terms (using project 2 resize).

Break Time!

The seam carving algorithm doesn't work so well with faces.

Break Time!

But it does appreciate fine art!



Break Time!

We'll start again in one minute.

Demo: Person_birthday

- Let's create a function that updates a person's age when they have a birthday

```
// MODIFIES: p
// EFFECTS:  Increases the person's age
//           by one. If they are now
//           older than 70, they are no
//           longer a ninja.
void Person_birthday(Person p) {

  // Implementation goes here

}
```

Note: This is doomed for failure…why?

Solution: Person_birthday

- We have to pass using a pointer or pass-by-reference in order to avoid the copy.

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  (*p).age += 1;
  if ((*p).age > 70) {
    (*p).isNinja = false;
  }
}
```

```
void Person_birthday(Person &p) {
  p.age += 1;
  if (p.age > 70) {
    p.isNinja = false;
  }
}
```

The Arrow Operator

- Use the -> operator as a shortcut for member access through a pointer.

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  (*p).age += 1;
  if ((*p).age > 70) {
    (*p).isNinja = false;
  }
}
```

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  p->age += 1;
  if (p->age > 70) {
    p->isNinja = false;
  }
}
```

Passing `structs` as parameters

- You **usually don't** want to pass by value. Usually don't need to **copy** the struct.[1]

```
void func(Person p);
```

- If you intend to **modify** the outside object, pass by **pointer or reference**.

```
void func(Person *p);
void func(Person &p);
```

- Otherwise, pass by **pointer-to-const** or **reference-to-const**. (Safer and more efficient than by value.)

```
void func(Person const *p);
void func(Person const &p);
```

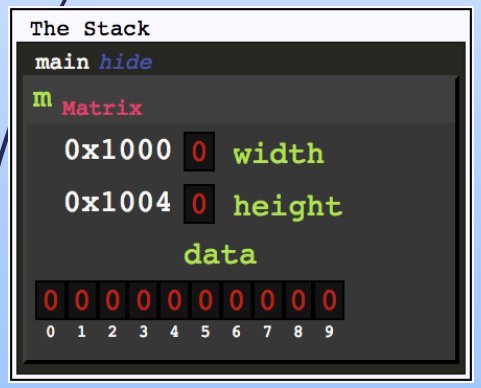
1 If the `struct` is very small, pass-by-value is acceptable.

Composing Data Types

- Example: Array of `Person`

```
Person people[3];
```

- Example: `Matrix` `struct` containing array




```
struct Matrix {
  int width;
  int height;
  int data[10];
};
...
Matrix m;
```

Exercise: Composing data types

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  int x;
  Person alex;
  alex.age = 20;
  Person jon;

  Person *people[] = {
    &alex, &jon
  };

  // print Alex's age
  cout << _____________ ;
}
```

Question

Which line(s) of code can go in the blank to print Alex's age?

A) *people.age

B) *(people->age)

C) people[0].age

D) people[0]->age