References and argument passing

Michael R. Nowak
University of Illinois Urbana-Champaign

Lesson Outline

1. Initialization of objects and function calls
2. References
3. Pass by reference
4. When do we pass arguments by reference?
5. Guidance for passing arguments

Lesson Outline

1. **Initialization of objects and function calls**
2. References
3. Pass by reference
4. When do we pass arguments by reference?
5. Guidance for passing arguments

Initialization of objects and function calls

Each time a function is called, its parameters are defined and initialized by the arguments passed in the function call

The semantics of argument passing are identical to the semantics of initialization

```
int add(int a, int b) {return a + b;}

int main() {
  int i = 4;
  int j = 6;
  int k = add(i,j);
}
```

Initialization of objects and function calls

```
int add(int a, int b) {return a + b;}

int main() {
  int i = 4;
  int j = 6;
  int k = add(i,j);
}
```

When the parameter is an object, we say that the argument is passed by value because a value is copied into the parameter object for initialization

Changes that the function makes to the parameter will never be reflected in the argument used to initialize that parameter: we're merely working with a copy of the argument

Now, consider the following:

```
int main() {
  int i = 4;
  int j = 6;

  int tmp = i;
  i = j;
  j = tmp;

  std::cout << "i = " << i << "; j = " << j << std::endl;
}
```

I would like to write a swap function that would perform the interchange of a and b's values

Now, consider the following:

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int i = 4;
  int j = 6;
  swap(i,j);
  std::cout << "i = " << i << "; j = " << j << std::endl;
}
```

The semantics of argument passing are identical to the semantics of initialization

Now, consider the following:

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int i = 4;
  int j = 6;
  swap(i,j);
  std::cout << "i = " << i << "; j = " << j << std::endl;
}
```

Can a function change the values of the actual arguments passed to it? That is, can we bind names within the scope of our function to objects that reside outside of it?

Lesson Outline

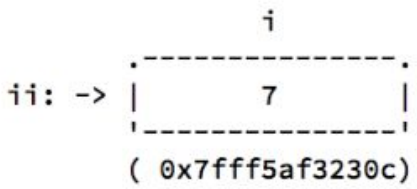
1. Initialization of objects and function calls
2. **References**
3. Pass by reference
4. When do we pass arguments by reference?
5. Guidance for passing arguments

References

A reference creates an alias or nickname for an object

We can bind the name `ii` to an `int` object `i` as follows:

```
int i  = 4;
int& ii = i; // ii becomes another name for i
```



We can also create a name that provides read-only access:

```
int i  = 4;
const int& ii = i; // ii affords read-only access to the underlying object
```

References

A reference cannot be rebound to some other object; because of this, all references must be initialized

```
int& a; // error: declaration of reference variable 'a' requires an initializer int &a
```

We cannot bind a reference to a literal:

```
int& a = 11; // error
```

however, we are allowed to take a reference to const to it:

```
const int& a = 11; // ok
```

Lesson Outline

1. Initialization of objects and function calls
2. References
3. **Pass by reference**
4. When do we pass arguments by reference?
5. Guidance for passing arguments

Reference parameters and function calls

Each time a function is called, its parameters are defined and initialized by the arguments passed in the function call

The semantics of argument passing are identical to the semantics of initialization

```
void swap(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int i = 4;
  int j = 6;
  swap(i,j);
  std::cout << "i == " << i << "; j == " << j << std::endl;
}
```

Reference parameters and function calls

```
void swap(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

When the parameter is a reference, we say that the argument is passed by reference because the parameter becomes an alias for the actual argument

The reference parameters are names within the scope of our function for the actual arguments residing outside of it (why is lifetime important?)

Changes that the function “make” on a reference parameter will always be reflected in the object bound to that reference

Lesson Outline

1. Initialization of objects and function calls
2. References
3. Pass by reference
4. **When do we pass arguments by reference?**
5. Guidance for passing arguments

When do we pass arguments by reference?

When we would like to:

- modify the arguments to which they are bound within our function
  - we have already seen an example of this with `swap`

- avoid copies
  - some objects in our program can become very large, while other objects (such as the IO types) cannot be copied

Avoiding copies

Consider the differences between the following function definitions

A

```
// passes vect by value
void f(std::vector<int> vect) {
  for (int v : vect)
    std::cout << v << ',';
  std::cout << std::endl;
}
```

B

```
// passes vect by reference
void f(std::vector<int>& vect) {
  for (int v : vect)
    std::cout << v << ',';
  std::cout << std::endl;
}
```

When called from `main` as

```
std::vector<int> numbers{1,2,3,4};
f(numbers);
```

Avoiding copies

```
// passes vect by value
void f(std::vector<int> vect) {
  for (int v : vect)
    std::cout << v << ',';
  std::cout << std::endl;
}
/*~*/
std::vector<int> numbers{1,2,3,4};
f(numbers);
```

Avoiding copies

```
// passes vect by reference
void f(std::vector<int>& vect) {
  for (int v : vect)
    std::cout << v << ',';
  std::cout << std::endl;
}
/*~*/
std::vector<int> numbers{1,2,3,4};
f(numbers);
```

Passing arguments by “vanilla” reference should generally be avoided

- Passing arguments by “vanilla” reference should generally be avoided (unless you have reason to use them)
  - They can lead to obscure bugs when you forget which actual arguments are changed through the parameters
- Instead, you should use **reference to const** when you are passing large objects and would like the **benefits of pass-by-reference, but do not need to modify the arguments**

```
void f(const std::vector<int>& vect) {
  // do something
}
```

Reference to const and function calls

What if we inadvertently wrote an unintended assignment to `vect.at(1)` in our trivial definition of `f()`, a function that was only supposed to read `vect`, not write to it?

```
void f(std::vector<int>& vect) {
     for (int v : vect)
        std::cout << v << ',';
    std::cout << std::endl;
    vect.at(1) = 20; // oops...
}
```

```
void f(const std::vector<int>& vect) {
     for (int v : vect)
        std::cout << v << ',';
    std::cout << std::endl;
    vect.at(1) = 20; // compiler error
}
```

Lesson Outline

1. Initialization of objects and function calls
2. References
3. Pass by reference
4. When do we pass arguments by reference?
5. **Guidance for passing arguments**

Guidance for passing arguments

- Use pass by value for small objects, such as the primitive built-in types
  - A simple copy of a small object is effective; don't pass by reference *unless* you have the reason to do so

- Use pass by reference to const for large objects, such as `std::vector`s and `std::string`s when you have no need to modify the object

- Use "vanilla" pass by reference only when needed
  - Favor returning a result rather than modifying an object through a reference argument

Is the following pass by value or reference?

```
void increment(int* p) {
     *p += 1;
}

int count = 11;
increment(&count);
```

Why ***should*** we prefer “passing by address” if the function is going to modify the actual argument opposed to vanilla pass-by-reference?