601.220 Intermediate Programming

C++ references

C++ references

Reference variable is an *alias*, another name for an existing variable (memory location)

- Used in many situations where pointers would be used in C
- References have restrictions that make them safer:
  - Can't be NULL
  - Must be initialized immediately
  - Once set to alias a variable, can't later be set to alias another

C++ references

To declare a reference of type `int`, use `int&`

- The & comes after the type
- Might remind you of the "address of" operator, but it's not the same

C++ references

References provide pointer-like functionality while hiding the "raw" pointers themselves

```
// ref1.cpp:
#include <iostream>

int main() {
    int i = 1;
    int *j = &i;
    std::cout << "i=" << i << ", *j=" << *j << std::endl;

    i = 9;
    std::cout << "i=" << i << ", *j=" << *j << std::endl;
    return 0;
}
```

```
$ g++ -c ref1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref1 ref1.o
$ ./ref1
i=1, *j=1
i=9, *j=9
```

C++ references

```cpp
// ref2.cpp:
#include <iostream>

int main() {
    int i = 1;
    int& j = i;
    std::cout << "i=" << i << ", j=" << j << std::endl;

    i = 9;
    std::cout << "i=" << i << ", j=" << j << std::endl;
    return 0;
}
```

```
$ g++ -c ref2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref2 ref2.o
$ ./ref2
i=1, j=1
i=9, j=9
```

C++ references

```cpp
// ref3.cpp:
#include <iostream>

int main() {
    int a = 5;
    int& b = a;
    // now b is "just another name for" a
    int* c = &a;
    // c is a "pointer" pointing to a
    std::cout << "&a=" << &a << std::endl;
    std::cout << "&b=" << &b << std::endl;
    std::cout << "&c=" << &c << std::endl;
    std::cout << "c=" << c << std::endl;
    return 0;
}
```

```
$ g++ -c ref3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref3 ref3.o
$ ./ref3
&a=0x7ffe95cdc9e4
&b=0x7ffe95cdc9e4
&c=0x7ffe95cdc9e8
c=0x7ffe95cdc9e4
```

C++ references

Function parameters with reference type are passed "by reference" – like passing "by pointer" but without the extra syntax inside the function

```
// if you have int a = 1, b = 2;  then call
// like this: swap(a, b) -- no ampersands!
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

C++ references

Symbols (Scope)	Values
a (main)	9
b (main)	1

```
// ref5.cpp:
#include <iostream>
using std::cout;
using std::endl;

void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int a = 1, b = 9;
    swap(a, b);
    cout << "a=" << a << ", b=" << b << endl;
    return 0;
}
```

```
$ g++ -c ref5.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref5 ref5.o
$ ./ref5
a=9, b=1
```

C++ references

Recall this example; `ch` passed by reference to `cin.get(char&)`

```
// ref6.cpp:
#include <iostream>
#include <cctype>

using std::cout;
using std::cin;

int main() {
    char ch;
    // read standard input char by char
    while(cin.get(ch)) { // pass ch by reference!
        cout << toupper(ch);
    }
    cout << endl;
    return 0;
}
```

C++ references

C++ has *both* pass by value (non-reference parameters) *and* pass by reference (reference parameters)

Function can have a mix of pass-by-value and pass-by-reference parameters

C++ references

```cpp
// ref7.cpp:
#include <iostream>

using std::cout;
using std::endl;

// `int a` and `int b` are passed *by value*
// `int& quo` and `int& rem` are passed *by reference*
void divmod(int a, int b, int& quo, int& rem) {
    quo = a / b;
    rem = a % b;
}

int main() {
    int a = 10, b = 3, quo, rem;
    divmod(a, b, quo, rem);
    cout << "a=" << a << ", b=" << b
         << ", quo=" << quo << ", rem=" << rem << endl;
    return 0;
}
```

```
$ g++ -c ref7.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref7 ref7.o
$ ./ref7
a=10, b=3, quo=3, rem=1
```

C++ references

Unfortunately, looking at the call itself doesn't tell you which parameters are passed by value and which are passed by reference:

```
divmod(a, b, quo, rem); // ???
```

Rather, you have to go look at the callee's parameter types:

```
void divmod(int a, int b, int& quo, int& rem) {
    ...
}
```

C++ references

C++ also has pointers, so you can still use the pass-by-pointer "workaround":

```
// this is still OK
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

// this is still OK
int a = 1, b = 2;
swap(&a, &b);
```

C++ references

Can we return a reference? Yes:

```
// ref9.cpp:
#include <iostream>

using std::cout;
using std::endl;

// Return reference to minimum argument
int& minref(int& a, int& b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int a = 5, b = 10;
    int& min = minref(a, b);
    min = 12;
    cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
}
```

Symbols (Scope)	Values
a (main), min (main)	12
b (main)	10

C++ references

```
$ g++ -c ref9.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref9 ref9.o
$ ./ref9
a=12, b=10, min=12
```

`minref` returns a reference to `int a`. When we later assign `min = 12`, we change both `min` and `a`.

What if we make `minref`'s arguments non-references?

C++ references

```cpp
// ref10.cpp:
#include <iostream>
using std::cout;
using std::endl;
int& minref(int a, int b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}
int main() {
    int a = 5, b = 10;
    int& min = minref(a, b);
    min = 6;
    cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
}
```

```
$ g++ -c ref10.cpp -std=c++11 -pedantic -Wall -Wextra
ref10.cpp:10:16: warning: reference to stack memory
associated with local variable 'a' returned [-Wreturn-stack-address]
        return a;
               ^
ref10.cpp:12:16: warning: reference to stack memory
associated with local variable 'b' returned [-Wreturn-stack-address]
        return b;
```

C++ references

Returning a reference to a local variable is just as bad as returning a pointer to one. In our original `minref` function, we avoided this by making the parameters themselves references.

```
int& minref(int& a, int& b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}
```

C++ references

Once a reference is set to alias a variable, it cannot later be set to alias another variable

Let's see an example

C++ references

```cpp
// ref11.cpp:
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int a = 5, b = 10;
    int& c = a;
    cout << "a=" << a << ", c=" << c << endl;
    c = b;
    cout << "a=" << a << ", c=" << c << endl;
    return 0;
}
```

```
$ g++ -c ref11.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref11 ref11.o
$ ./ref11
a=5, c=5
a=10, c=10
```

`c = b` assigns b's value (10) to c (and therefore also to a)

C++ references

A reference variable must be initialized immediately.

```
// ref12.cpp:
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int& a;
    int b = 10;
    a = b;
    cout << a << endl;
    return 0;
}
```

```
$ g++ -c ref12.cpp -std=c++11 -pedantic -Wall -Wextra
ref12.cpp:7:10: error: declaration of reference variable
                            'a' requires an initializer
    int& a;
         ^
1 error generated.
```

C++ references

A reference cannot be NULL

```
// ref13.cpp:
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int& a = NULL;
    if(a == NULL) {
        cout << "a is NULL" << endl;
    }
    return 0;
}
```

```
$ g++ -c ref13.cpp -std=c++11 -pedantic -Wall -Wextra
ref13.cpp:7:10: error: non-const lvalue reference to type 'int'
                           cannot bind to a temporary of type 'long'
    int& a = NULL;
         ^   ~~~~
1 error generated.
```

C++ references

A reference can be `const` – if so, can't subsequently assign via that reference

. . . but you can still assign to the original non-const variable, or via a non-const reference to it

C++ references

```cpp
// ref14.cpp:
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int a = 1;
    int& b = a;
    const int& c = a;
    a = 2;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    b = 3;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    c = 4;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    return 0;
}
```

```
$ g++ -c ref14.cpp -std=c++11 -pedantic -Wall -Wextra
ref14.cpp:15:7: error: read-only variable is not assignable
    c = 4;
    ~ ^
1 error generated.
```

C++ references

```cpp
// ref15.cpp:
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int a = 1;
    int& b = a;
    const int& c = a;
    a = 2;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    b = 3;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    //c = 4;
    //cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    return 0;
}
```

```
$ g++ -c ref15.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref15 ref15.o
$ ./ref15
a=2, b=2, c=2
a=3, b=3, c=3
```

C++ references

We've seen the difference between pass by reference and pass by value

In C++, when passing objects, we generally pass by reference

- `const` reference if modification is not permitted
- Normal reference otherwise

Quiz!

What is the output of the following program?

```
#include <iostream>
void times3(int& x) {
  x *= 3;
}

int main() {
  int a = 2;
  int b = a;
  int& c = a;
  times3(a);
  times3(b);
  times3(c);
  std::cout << a << ", " << b << ", "
            << c << std::endl;
  return 0;
}
```

A. `6, 2, 6`

B. `6, 6, 6`

C. `6, 18, 18`

D. `18, 6, 18`

E. The program doesn't compile

Quiz - answer

What is the output of the following program?

```
#include <iostream>
void times3(int& x) {
  x *= 3;
}

int main() {
  int a = 2;
  int b = a;
  int& c = a;
  times3(a);
  times3(b);
  times3(c);
  std::cout << a << ", " << b << ", "
            << c << std::endl;
  return 0;
}
```

Symbols (Scope)	Values
a (main), c (main)	18
b (main)	6

C++ objects: passing by reference

Question: What's the difference between passing by `const` reference and passing by value?

```
int sum(vector<int> vec) { ... };

int sum(const vector<int>& vec) { ... };
```

C++ objects: passing by reference

Question: What's the difference between passing by `const` reference and passing by value?

```
int sum(vector<int> vec) { ... };

int sum(const vector<int>& vec) { ... };
```

First form creates a copy, second form doesn't.

Essentially no downside, which is one big reason we usually pass class objects by reference

Another reason is related to dynamic binding, as we'll see later

C++ references

You should be able to tell the differences among below funcs:

```
void func(int a, int b);

void func(const int a, const int b);

void func(int& a, int& b);

void func(const int& a, const int& b);

void func(int* a, int* b);

void func(const int* a , const int* b);
```