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:
  - Cannot be `NULL`
  - Must be initialized immediately
  - Once set to alias a variable, cannot 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 is 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++ -o ref1 ref1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./ref1
```

```
i=1, *j=1
i=9, *j=9
```

C++ references

```
// ref2.cpp
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      int& j = i;
6      std::cout << "i=" << i << ", j=" << j << std::endl;
7
8      i = 9;
9      std::cout << "i=" << i << ", j=" << j << std::endl;
10     return 0;
11 }
```

```
$ g++ -o ref2 ref2.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./ref2
```

```
i=1, j=1
i=9, j=9
```

C++ references

```
// 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" Opointing 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++ -o ref3 ref3.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./ref3
&a=0x7fffc4ddc7c4
&b=0x7fffc4ddc7c4
&c=0x7fffc4ddc7c8
c=0x7fffc4ddc7c4
```

C++ references

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

// ref4.cpp

```
1  // if you have int a = 1, b = 2;  then call
2  // like this: swap(a, b) -- no ampersands!
3  void swap(int& a, int& b) {
4      int tmp = a;
5      a = b;
6      b = tmp;
7  }
```

C++ references

At line 11:

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

```
// ref5.cpp

1  #include <iostream>
2
3  void swap(int& a, int& b) {
4      int tmp = a;
5      a = b;
6      b = tmp;
7  }
8  int main() {
9      int a = 1, b = 9;
10     swap(a, b);
11     std::cout << "a=" << a << ", b=" << b << std::endl;
12     return 0;
13 }

$ g++ -o ref5 ref5.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./ref5

a=9, b=1
```

C++ references

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

```
// ref6.cpp

#include <iostream>
#include <cctype>

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

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

// `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);
    std::cout << "a=" << a << ", b=" << b
        << ", quo=" << quo << ", rem=" << rem << std::endl;
        return 0;
}
```

```
$ g++ -o ref7 ref7.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./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*:

// ref8.cpp

```
1  // this is still OK
2  void swap(int *a, int *b) {
3      int tmp = *a;
4      *a = *b;
5      *b = tmp;
6  }
7
8  // this is still OK
9  int a = 1, b = 2;
10 swap(&a, &b);
```

C++ references

Can we return a reference? Yes

At line 16:

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

```
// ref9.cpp

#include <iostream>

// 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;
    std::cout << "a=" << a << ", b=" << b << ", min=" << min << std::endl;
}
```

C++ references

```
$ g++ -o ref9 ref9.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./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

```
    // ref10.cpp

 1  #include <iostream>
 2
 3  int& minref(int a, int b) {
 4      if(a < b) {
 5          return a;
 6      } else {
 7          return b;
 8      }
 9  }
10
11  int main() {
12      int a = 5, b = 10;
13      int& min = minref(a, b);
14      min = 6;
15      std::cout << "a=" << a << ", b=" << b << ", min=" << min << std::endl;
16  }

    $ g++ -o ref10 ref10.cpp -std=c++11 -pedantic -Wall -Wextra

    ref10.cpp: In function int& minref(int, int):
    ref10.cpp:3:17: warning: reference to local variable a returned [-Wreturn-local-addr]
     int& minref(int a, int b) {
                 ~~~~~^
    ref10.cpp:3:24: warning: reference to local variable b returned [-Wreturn-local-addr]
     int& minref(int a, int 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>

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

```
$ g++ -o ref11 ref11.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./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>

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

```
$ g++ -o ref12 ref12.cpp -std=c++11 -pedantic -Wall -Wextra

ref12.cpp: In function int main():
ref12.cpp:4:10: error: a declared as reference but not initialized
     int& a;
          ^
```

C++ references

A reference cannot be NULL

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

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

```
$ g++ -o ref13 ref13.cpp -std=c++11 -pedantic -Wall -Wextra
ref13.cpp: In function int main():
ref13.cpp:4:14: warning: converting to non-pointer type int from NULL [-Wconver
     int& a = NULL;
              ^~~~
In file included from /usr/include/_G_config.h:15,
                 from /usr/include/libio.h:31,
                 from /usr/include/stdio.h:74,
                 from /usr/include/c++/8/cstdio:42,
                 from /usr/include/c++/8/ext/string_conversions.h:43,
                 from /usr/include/c++/8/bits/basic_string.h:6400
```

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

```
// ref14.cpp

#include <iostream>

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

```
$ g++ -o ref14 ref14.cpp -std=c++11 -pedantic -Wall -Wextra

ref14.cpp: In function int main():
ref14.cpp:11:13: error: assignment of read-only reference c
         c = 4;
             ^
```

C++ references

```
// ref15.cpp

#include <iostream>

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

```
$ g++ -o ref15 ref15.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./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?

```
1  #include <iostream>
2  void times3(int& x) {
3      x *= 3;
4  }
5
6  int main() {
7      int a = 2;
8      int b = a;
9      int& c = a;
10     times3(a);
11     times3(b);
12     times3(c);
13     std::cout << a << ", " << b << ", " << c << std::endl;
14     return 0;
15 }
```

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?

```
1  #include <iostream>
2  void times3(int& x) {
3      x *= 3;
4  }
5
6  int main() {
7      int a = 2;
8      int b = a;
9      int& c = a;
10     times3(a);
11     times3(b);
12     times3(c);
13     std::cout << a << ", " << b << ", " << c << std::endl;
14     return 0;
15 }
```

At line 13:

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) { ... };
```

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 functions:

```
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);
```