University of Southern California

Viterbi School of Engineering

SW for EE

References

References: Professor Mark Redekopp’s slide units, online resources (papers, articles, etc.

Swap Two Variables

- Classic example of issues with local variables:
  - Write a function to swap two variables
- Pass-by-value doesn’t work
  - Copy is made of x,y from main and passed to x,y of swapit…Swap is performed on the copies
- Pass-by-reference (pointers) does work
  - Addresses of the actual x,y variables in main are passed
  - Use those address to change those physical memory locations

```
int main()
{ int x=5,y=7;
  swapit(x,y);
  cout <<“x,y=“ << x << “,” <<
y << endl;
}

void swapit(int x, int y)
{  int temp;
   temp = x;
   x = y;
   y = temp;
}
```

program output: x=5,y=7

```
int main()
{ int x=5,y=7;
  swapit(&x,&y);
  cout <<“x,y=“ << x << “,” <<
y << endl;
}

void swapit(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

program output: x=7,y=5

C++ Reference Variables

- So you want a function to actually modify a variable from another function but you don’t like pointers and they confuse you?
  - Did you know that everyday many pointers are left pointing to NULL? humanity leaked :D
  - You may instead use C++ Reference variables
- C++ reference variables essentially pass arguments via pointer/address but use the syntax of pass-by-value, i.e., no more de-referencing
  - Questions: what syntax are we referring to?

Using C++ Reference Variables

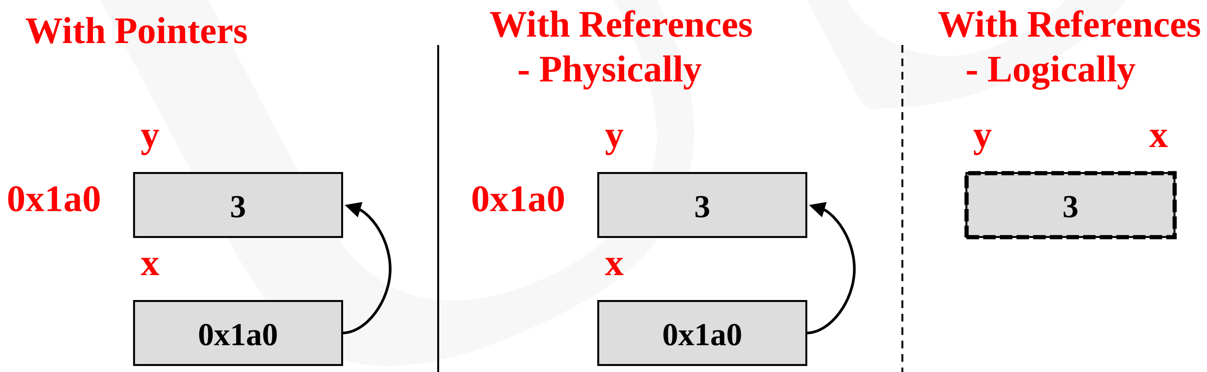
- To declare a reference variable, use the ‘&’ operator in a *declaration!*
  - Poor choice by C++ because it is confusing since ‘&’ is already used for the ‘address of operator’ when used in an expression (i.e. non-declaration)
- Behind the scenes the compiler will essentially access variable with a pointer
- But you get to access it like a normal variable without dereferencing
- Think of a reference variable as an alias

```
int main()
{
  int y = 3;
  doit(&y); //address-of oper.
  cout << y << endl;
  return 0;
}

int doit(int *x)
{
   *x = *x - 1;
   return *x;
}
```

Using pointers

```
int main()
{
  int y = 3;
  doit(y);
  cout << y << endl;
  return 0;
}

int doit(int &x) // Ref. dec.
{
   x = x - 1;
   return x;
}
```

Using C++ References
Output: ‘2’ in both programs


With Pointers

With References - Physically

With References - Logically

Swap Two Variables

- Pass-by-value => Passes a copy
- Pass-by-reference =>
  - Pass-by-pointer/address => Passes address of actual variable
  - Pass-by-C++ Reference => Passes an alias to actual variable

```
int main()
{
  int x=5,y=7;
  swapit(x,y);
  cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

program output: x=5,y=7

```
int main()
{
  int x=5,y=7;
  swapit(&x,&y);
  cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

program output: x=7,y=5

```
int main()
{
  int x=5,y=7;
  swapit(x,y);
 cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

program output: x=7,y=5

When to Use References

- Whenever you want to actually modify an input parameter/argument, i.e., a local variable from another function
- Great for passing big struct or class objects
    - Because no copy will be made, (pass-by-value would have wasted time copying contents to new memory)

```
class GradeBook{
 public:
  int grades[8][100];
};

int main()
{

  GradeBook gb;
  ...
  double average = process_it(gb);
  return 0;
}
double process_it(GradeBook &mygb)
{
   double sum = 0;
   for(int i=0; i < 8; i++)
     for(int j=0; j < 100; j++)
       sum += mygb.grades[i][j];


   mygb.grades[0][0] = 91;


   sum /= (8*100);

   return sum;
}
```

Const arguments

- An aside:
  - If we want an extra safety precaution for our own mistakes, we can declare arguments as 'const'
  - The compiler will produce an error to tell you that you have written code that will modify the object you said should be constant
  - Doesn’t protect against back-doors like pointers that somehow point at these data objects

```
class GradeBook{
 public:
  int grades[8][100];
};

int main()
{

  GradeBook gb;
  ...
  double average = process_it(gb);
  return 0;
}
double process_it(const GradeBook &mygb)
{
   double sum = 0;
   for(int i=0; i < 8; i++)
     for(int j=0; j < 100; j++)
       sum += mygb.grades[i][j];

   mygb.grades[0][0]  = 91;
   // modification of mygb
   // compiler will produce ERROR!


   sum /= (8*100);

   return sum;
}
```

Vector/Deque/String Suggestions

- When you pass a vector, deque, or even C++ string to a function a deep copy will be made which takes time
- Copies may be desirable in a situation to make sure the function alter your copy of the vector/deque/string
- But passing by const reference saves time and provide the same security

Will be discussed later

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> my_vec;
  for(int i=0; i < 5; i++){
    // my_vec[i] = i+50; // doesn’t work
    my_vec.push_back(i+50);
  }

  // can myvec be different upon return?
  do_something1(myvec);

  // can myvec be different upon return?
  do_something2(myvec);
  return 0;
}
void do_something1(vector<int> v)
{
  // process v;
}
void do_something2(const vector<int>& v)
{
  // process v;
}
```

Reference Gotchas!

- Returning a reference to a dead variable, i.e., a local variable of a function that just completed
- avg was a local variable and thus was deallocated when process_it completed

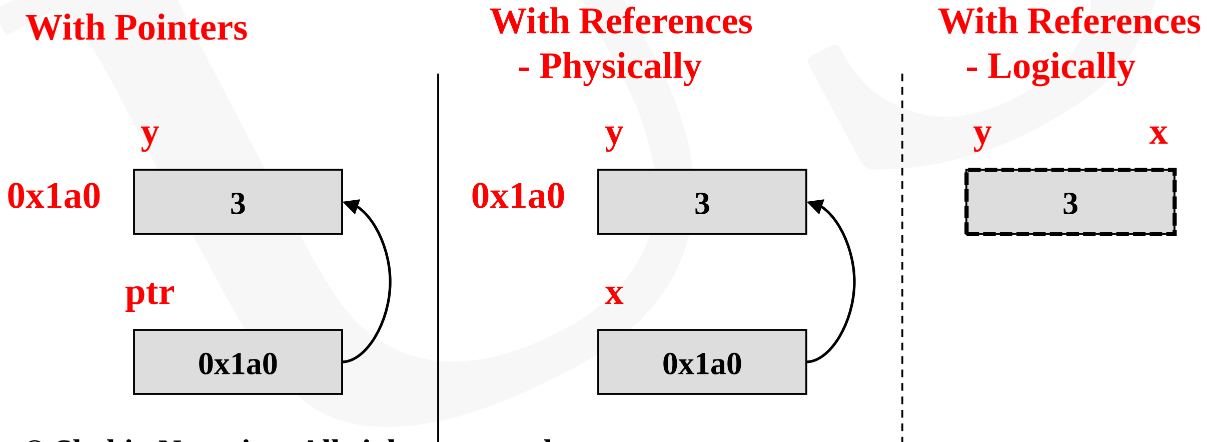
```
class GradeBook{
 public:
  int grades[8][100];
};

int main()
{

  GradeBook gb;
  double& average = process_it(gb);
  cout << “Avg: ” << average << endl;
  // Possible seg. fault / prog. crash
  return 0;
}
double &process_it(const GradeBook &mygb)
{
   double avg = 0;
   for(int i=0; i < 8; i++)
     for(int j=0; j < 100; j++)
       avg += mygb.grades[i][j];


   avg /= (8*100);

   return avg;  // reference to avg
                //  is returned...
}
```

Exercise: returnref

Using C++ References

- Mainly used for parameters, but can use it within the same function
- A variable declared with an ‘int &’ doesn’t store an int, but stores a reference/alias for an actual integer
- MUST assign to the reference variable when you declare it

```
int main()
{
  int y = 3, *ptr;
  ptr    = &y;  // address-of
                //  operator

 int &z;  // NO! must assign

 int &x = y;   // reference
                // declaration
  // we've not copied
  // y into x
  // we've created an alias

  x++;    // y just got incr.
  cout << y << endl;
  return 0;
}
```


With Pointers

With References - Physically

With References - Logically

Output: y=4 in both programs

Using C++ References

- To summarize, references are less powerful but safer than pointers
- It is not possible to refer directly to a reference object after it is defined; any occurrence of its name refers directly to the object it references
- Unlike pointers, once a reference is created, it cannot be reseated, i.e., a reference to an object cannot later be made to reference another object
- Unlike pointers, references must be initialized as soon as they are created, i.e., references cannot be uninitialized
- References to local and global variables must be initialized where they are defined
- References which are data members of class instances must be initialized in the initializer list of the class constructor
- References cannot be *null*, whereas pointers can; every reference refers to an object, however it may or may not be valid