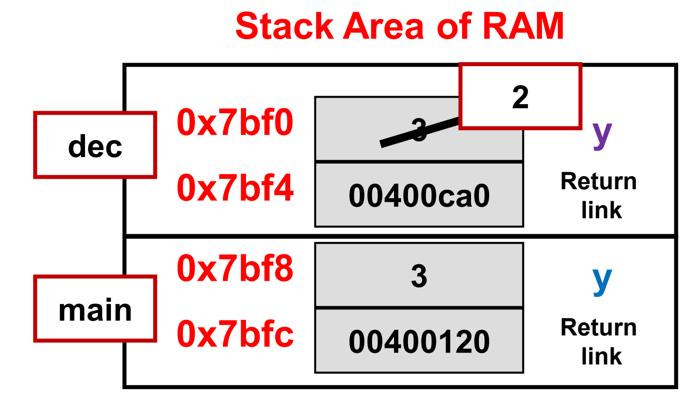
CS 103 Unit 4e – C++ References

Recall: Pass-by-Value

- Each function has its own memory on the the **system stack** where all data related to the function is stored including:
  - **Local variables**
  - **Arguments to the function**
  - Return link (where to return) to the calling code
- By default, parameters are **passed-by-value** (i.e. a copy is made)
- Thus, one function CANNOT modify the data of another
- Alternative: **Pass-by-reference** (pointers)

```
// Prototype
void dec(int);

int main()
{
  int y = 3;
  dec(y);
  cout << y << endl;  // prints ___
  return 0;
}

void dec(int y)
{
   y--;
}
```

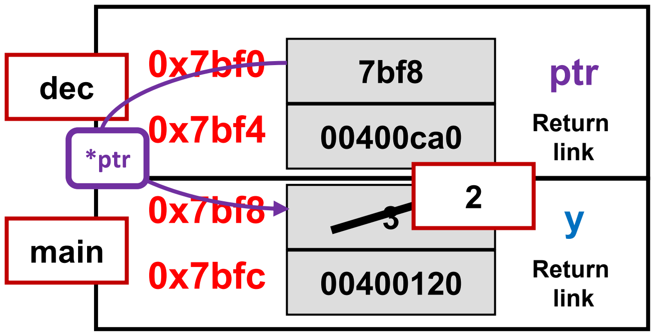


Pass-by-Reference (Using Pointers)

- To allow a function to modify the data of another, we learned we must pass pointers
- But that syntax is a bit confusing. Is there an easier way?
- **Yes!** C++ References
  - These are likely pointers behind the scenes but simplify the syntax and semantics

```
// Prototype
void dec(int);

int main()  // caller
{
  int y = 3;
  dec(&y);
  cout << y << endl; // prints 2
  return 0;
}

void dec(int* ptr)  // callee
{
   *ptr = *ptr - 1; // or (*ptr)--;
}
```

Stack Area of RAM

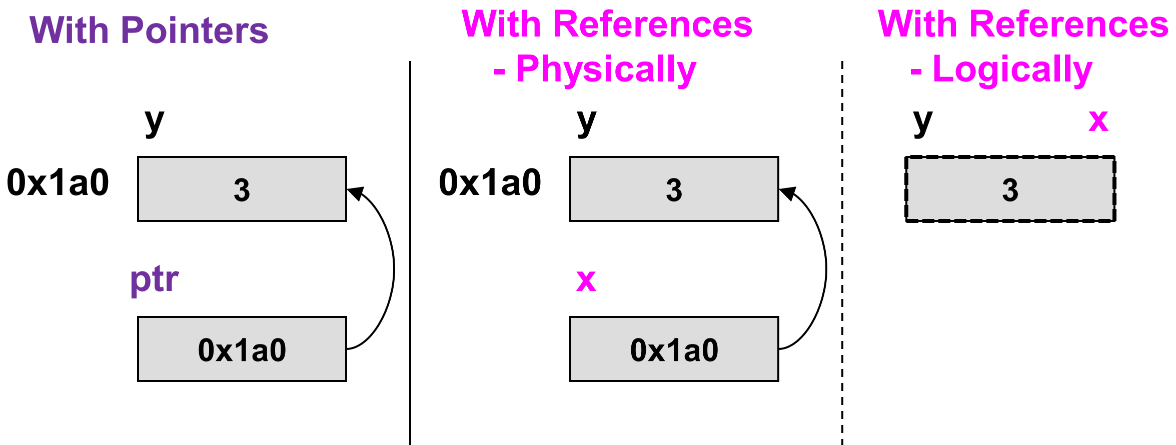


C++ Reference Variables

- So, you want to use pass-by-reference to allow a function to modify data from another, but you don't like pointers and they confuse you?
  - Too bad. Don't give up! You CAN understand pointers…keep working at it. And pointers are necessary in several contexts (dynamic allocation, etc.)
  - BUT…
  - You can also use C++ Reference variables in many contexts
- C++ reference variables essentially pass arguments via pointer/address behind the scenes but use a much simpler syntax (i.e. no more de-referencing)
  - We needed you to know what's actually happening behind the scenes. Thus, we taught you pointers.
  - But you can also use the simplified syntax with C++ references

Using C++ References

- Declaring a reference type (T&) creates an **alias** (alternate name) for another already-existing variable
  - **T&** is **NOT** another variable; does **NOT** require memory
  - Ex: **`int&`** doesn't store an int, but is an **alias** for an actual int variable
- Many people call references: "Syntactic sugar" (to make programmer's life easy) to avoid pointer syntax
- MUST assign to the reference variable when you declare it.
- **Most common usage: Passing parameters**



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

  // reference declaration
  int &x = y;

  // x is not another variable.
  // Rather it's an alias!
  // Doing ops on x really happens
  //   on y
  // Now x can never reference
  //   any other int…only y!

  x++;    // increment y

  cout << y << endl; // 4 prints

  int &z;     // NO! must assign

  int w = 5;
  x = w;  // doesn't make x
          // reference w...copies
          // w into y;
  return 0;
}
```

A New Way to Pass by Reference

- To declare a C++ reference, use the & symbol after the type in a ***declaration!***
  - Poor choice by C++ because & 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;
}

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

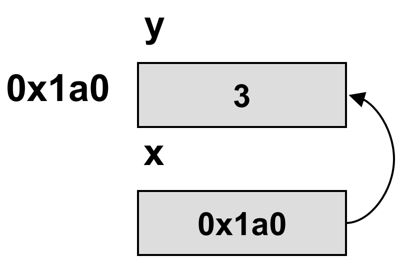
Using pointers

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

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

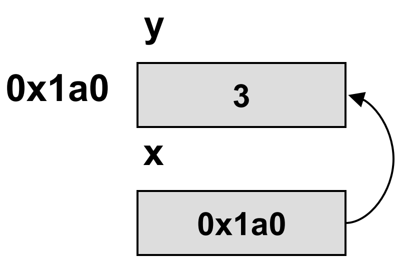
Using C++ References

Output: '2' in both programs

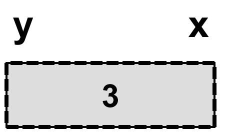
With Pointers



With References - Physically



With References - Logically



Kinds of References

Pointers

- A variable (like any other) which occupies memory and stores an address of another variable and can be updated (like any other variable) to store a new address to some other variable
- Declared with the `type*` syntax (e.g. `int*`, `char*`, `Item*`)

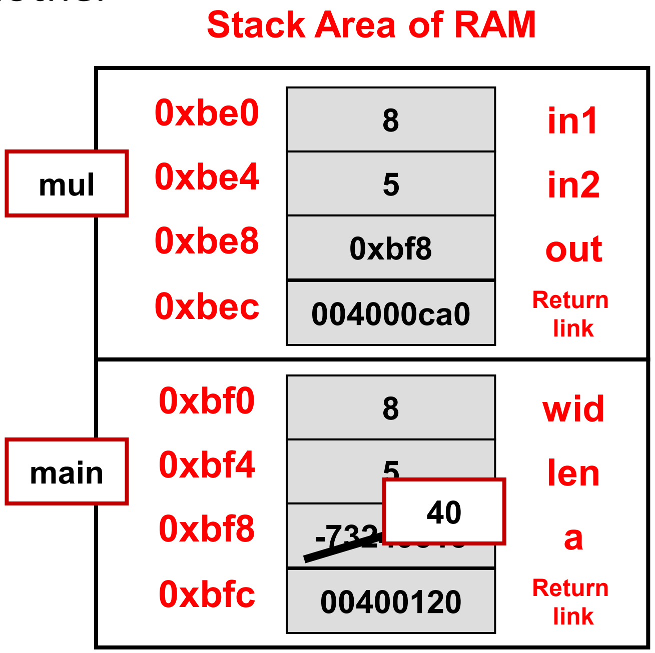
C++ Reference Variable

- A special declaration that simply gives a second (or third, or fourth) name to an already-declared variable
- Declared with the `type&` syntax (e.g. `int&`, `string&`, `Item&`)
- Does not occupy any memory (just tells the compiler to allow another name to reference some other variable)

Important Note: When we use the general term "reference" as in "pass-by-reference" we can use EITHER **pointers OR C++ Reference Variables.**

Correct Usage of Pointers

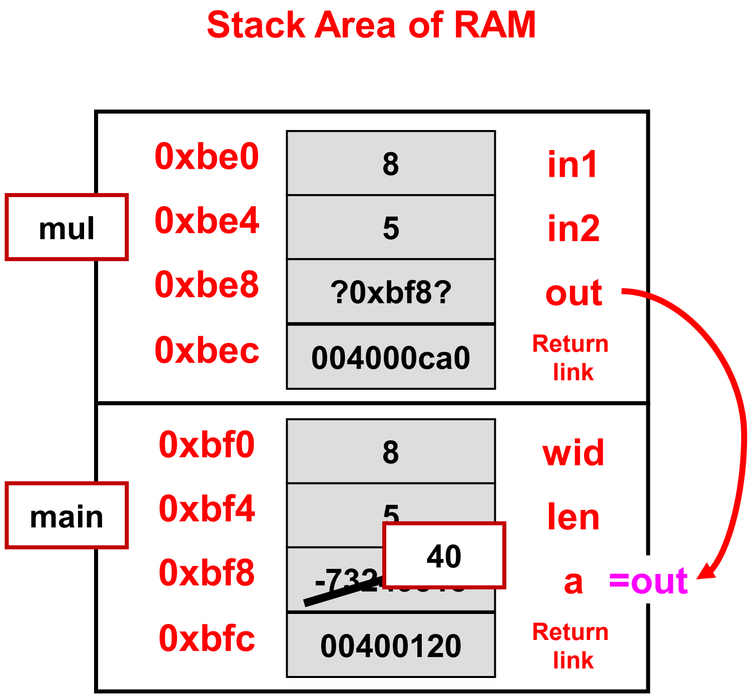
- Commonly functions will take some inputs and produce some outputs
  - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
  - We could use the return value from the function but let's practice with pointers
- Can use a pointer to have a function modify the variable of another



```
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);


int main()
{
  int wid = 8, len = 5, a;
  mul2(wid,len,&a);
  cout << "Ans. is " << a << endl;
  return 0;
}

int mul1(int in1, int in2)
{
  return in1 * in2;
}

void mul2(int in1, int in2, int* out)
{
  *out = in1 * in2;
}
```

Now with C++ References

- We can pass using C++ reference
- The reference 'out' is just an alias for 'a' back in main
  - In memory, it might actually be a pointer, but you don't have to dereference (the kind of stuff you have to do with pointers)

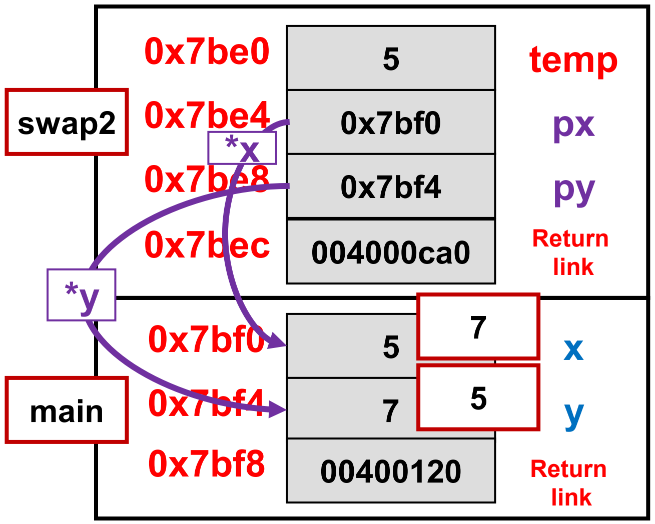


```
// Computes the product of in1 & in2
void mul(int in1, int in2, int& out);

int main()
{
  int wid = 8, len = 5, a;
  mul(wid,len,a);
  cout << "Ans. is " << a << endl;
  return 0;
}

void mul(int in1, int in2, int& out)
{
  out = in1 * in2;
}
```

Pass-by-Reference (Using Pointers)

- Classic example of issues with local variables:
  - Write a function to swap two variables
- 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

Stack Area of RAM

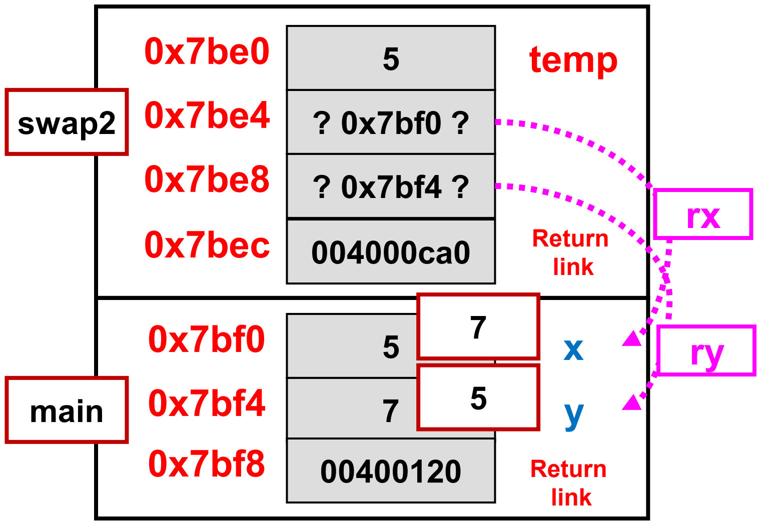


```
#include <iostream>
using namespace std;
void swap2(int* x, int* y);

int main()
{
  int x=5,y=7;
  swap2(&x, &y);
  cout << " x=" << x;
  cout << " y=" << y << endl;
}
void swap2(int* px, int* py)
{
   int temp = *px;
   *px = *py;
   *py = temp;
}
```

Pass-by-Reference (C++ Reference)

- Classic example of issues with local variables:
  - Write a function to swap two variables
- Pass-by-reference with C++ References makes the syntax cleaner
  - Behind scenes it is likely still passing pointers

Stack Area of RAM



```
#include <iostream>
using namespace std;
void swap2(int& rx, int& ry);

int main()
{
  int x=5,y=7;
  swap2(x, y);
  cout << " x=" << x;
  cout << " y=" << y << endl;
}
void swap2(int& rx, int& ry)
{
   int temp = rx;
   rx = ry;
   ry = temp;
}
```

Swap Two Variables Summary

- 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;
}
```

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 *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Output: x=7,y=5

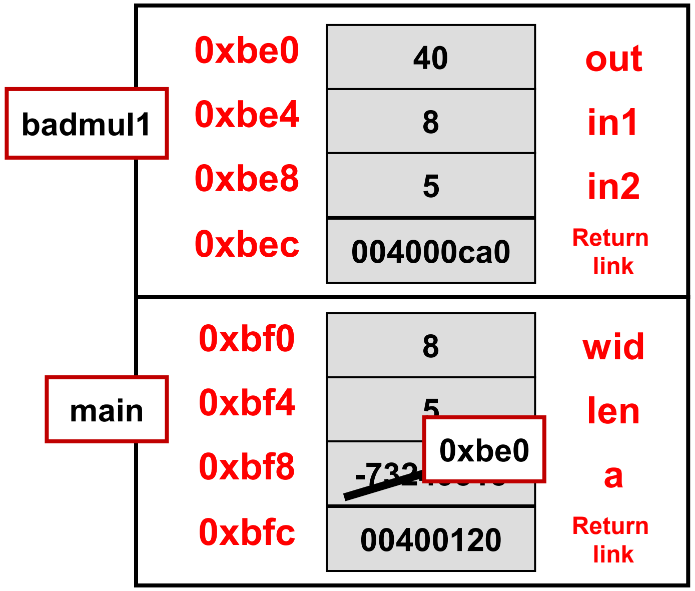
```
int main()
{
  int x=5,y=7;
  swapit(x,y);
  cout <<"x,y="<< x<<","<< y;
  cout << endl;
}

void swapit(int &rx, int &ry)
{
   int temp;
   temp = rx;
   rx = ry;
   ry = temp;
}
```

Output: x=7,y=5

Misuse of Pointers/References

- Make sure you don't return a pointer or reference to a dead variable
- You might get lucky and find that old value still there, but likely you won't

Stack Area of RAM

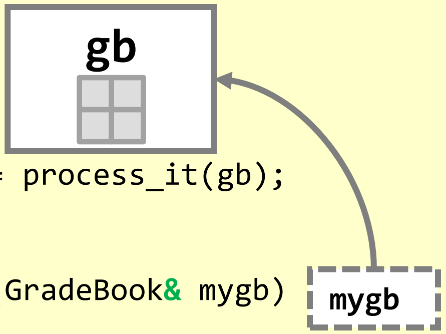


```
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);
int& badmul2(int in1, int in2);

int main()
{
  int wid = 8, len = 5;
  int *a = badmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  return 0;
}

// Bad! Returns a pointer to a var.
// that will go out of scope
int* badmul1(int in1, int in2)
{
  int out = in1 * in2;
  return &out;
}

// Bad! Returns a reference to a var.
// that will go out of scope
int& badmul1(int in1, int in2)
{
  int out = in1 * in2;
  return out;
}
```

When to Use References

- **Reason 1:** Whenever you want to actually **modify an input parameter/argument of the calling function** (e.g. modify a local variable from another function)
- **Reason 2:** To **avoid making a copy** when passing big struct or class objects
  - Because no copy will be made, (pass-by-value would have wasted time copying contents to new memory)

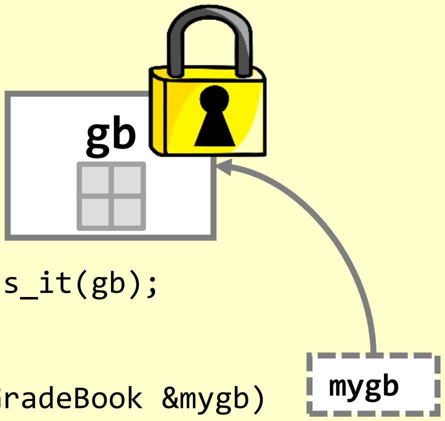
```
class GradeBook{
 public:
  int grades[8][100]; // Large amount of data
};

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 (compiler check only)

```
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 const Gradebook
   // compiler will produce an ERROR!


   sum /= (8*100);

   return sum;
}
```



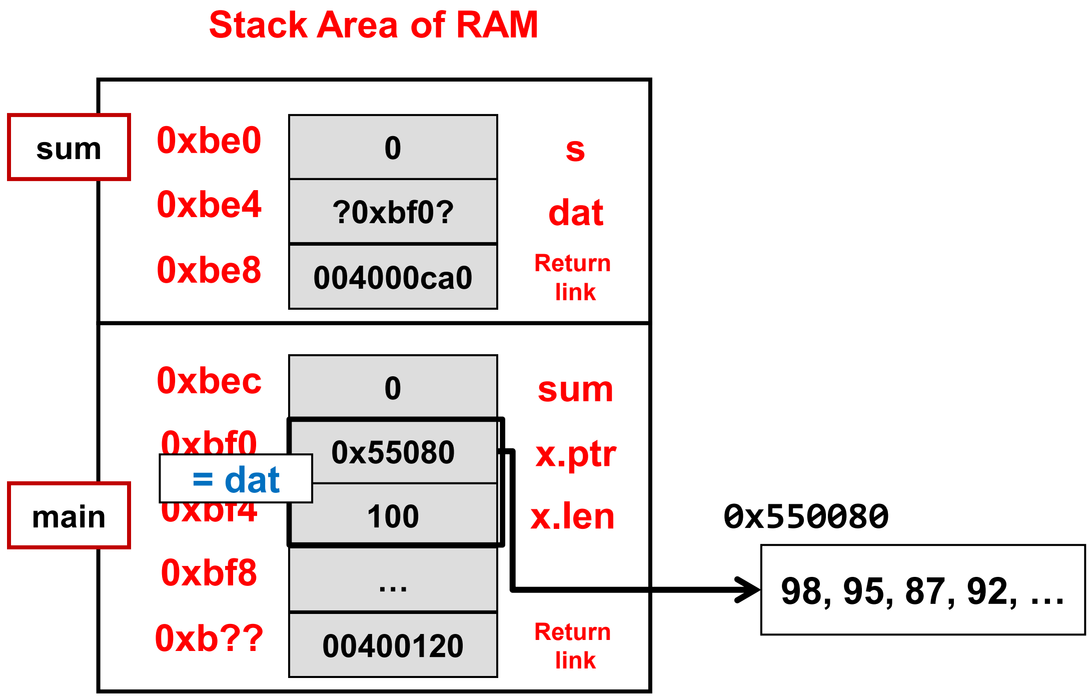
Vector/Deque/String Suggestions

- When you pass a vector, deque, or even C++ string to a function a deep copy will be made.
  - SLOW!!
- The advantage of a copy is that the function CANNOT alter the original vector/deque/string
- But passing by const reference ***saves time and provides the same security***.

```
#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; // recall 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;
}
```

Pass by Reference

- Notice no copy of x need be made since we pass it to sum() by reference
  - Notice that likely the computer passes the address to sum() but you should just think of dat as an alias for x
  - The const keyword tells the compiler to double check that we don't modify the vector (giving the safety of pass-by-value but the performance of pass-by reference)



```
// Computes the sum of a vector
int sum(const vector<int>&);

int main()
{
  int result;
  vector<int> x(100);

  // fill x w/ {98, 95, 87, 92, ...};

  result = sum(x);
}

int sum(const vector<int>& dat)
{
  int s = 0;
  for(size_t i=0; i < dat.size(); i++)
  {
     s += dat[i];
  }
  return s;
}
```

Pointers vs. References Summary

- How to tell references and pointers apart
  - Check if you see the '&' or '*' in a type declaration or expression

	With a Type	In an Expression
&	Indicates a C++ Reference Var (`int &val, vector<int> &vec`)	Address-of yields a pointer to the object Adds a * to the type of variable (`int x; int* p; p = &x;`)
*	Declares a pointer type variable (`int *valptr = &val, vector<int> *vecptr = &vec`)	De-Reference (Value @ address) Cancels a * from the type of variable (`int* p = new int; *p = 5;`)