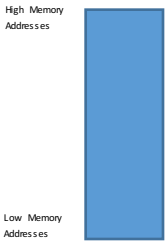
CS 31 : Introduction To Computer Science

Howard A. Stahl

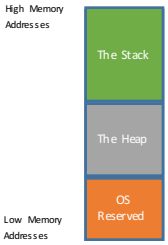
The RAM In Your Computer

- We Are Very Spoiled, Living In A World Of Lots Of Memory…



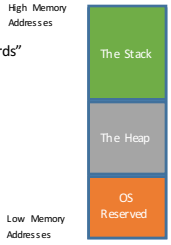
The RAM In Your Computer

- We Are Very Spoiled, Living In A World Of Lots Of Memory…



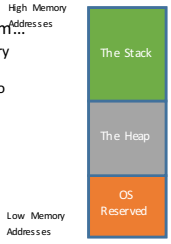
The RAM In Your Computer

- The Stack Is What We Have Been Using All Along…
  - It Holds Call Stack "Activation Records" With All The Declared "Automatic" Variables We Have Ever Made



The RAM In Your Computer

- The Heap Is What Were Our Dynamic Variables Are Come From…
  - Calls To `new` Offer Available Memory From The Heap
  - Calls To `delete` Return Memory To The Heap To Be "Recycled"

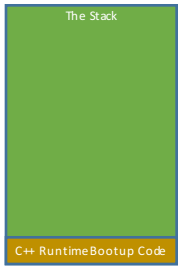


Let's Try Driving Some Code…

```
void foo( int i )
{
   int a=12;
   cout << i << a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```

Let's Try Driving Some Code…

```
void foo( int i )
{
   int a=12;
   cout << i << a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```

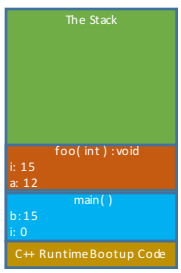


Let's Try Driving Some Code…

```
void foo( int i )
{
   int a=12;
   cout << i << a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```

Let's Try Driving Some Code…

```
void foo( int i )
{
   int a=12;
   cout << i << a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



Let's Try Driving Some Code…

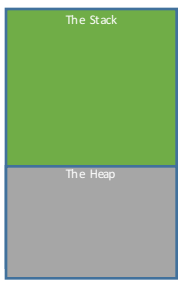
- Automatic Variables You Declared Are Managed By The Compiler For You
  - Declared Variables Are Placed Into An Activation Record On The Stack
  - Variables Live Within Their "Scope"
    - Are "Born" When Their Scope Comes Into View
    - And "Die Off" When Their Scopes Ends

Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```

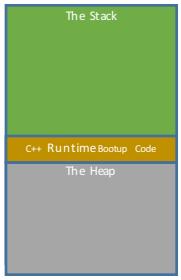
Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



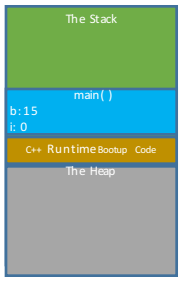
Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



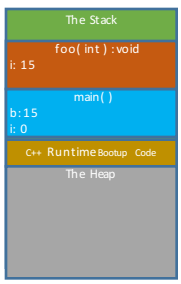
Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



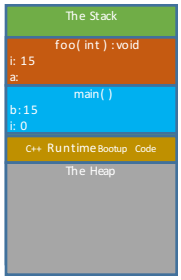
Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



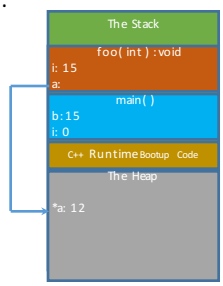
Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



Let's Try Driving Some Code…

```
void foo( int i )
{
   int* a = new int( 12 );
   cout << i << *a;
}

int main( )
{
   int b = 15;
   int i = 0;
   foo( b );
   return( 0 );
}
```



Let's Try Driving Some Code…

- Dynamic Variables Are Given An Address At Run-Time
  - Dynamic Variables Come From The Heap
  - Until You `delete` Them, Heap Addresses Are Unavailable To Any Other Process

Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
```

Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
```

Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
ptrInt = nullptr;
```

Introducing The Value nullptr

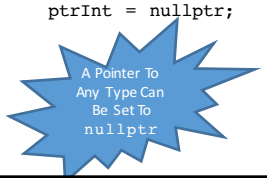
- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
ptrInt = nullptr;
```

Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

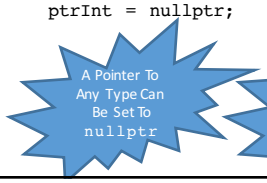
```
int * ptrInt;
ptrInt = nullptr;
```

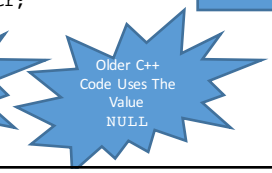


Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
ptrInt = nullptr;
```





Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
ptrInt = nullptr;
```

- A `nullptr` Value Can Be A Guard You Can Check For

Introducing The Value nullptr

- Recall When You Declare A Pointer, It Is Initially Unusable…

```
int * ptrInt;
ptrInt = nullptr;
```

- A `nullptr` Value Can Be A Guard You Can Check For

```
if (ptrInt != nullptr)
{    ....  // don't use *ptrInt here        }
else
{    ....  // use *ptrInt here safely       }
```

Let's Go Back To C-Strings!

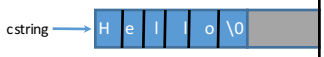
```
char cstring[80];
strcpy( cstring, "Hello" );
```

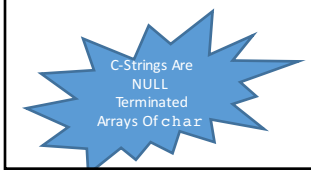
Let's Go Back To C-Strings!

```
char cstring[80];
strcpy( cstring, "Hello" );
```

Let's Go Back To C-Strings!

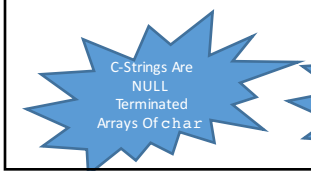
```
char cstring[80];
strcpy( cstring, "Hello" );
```

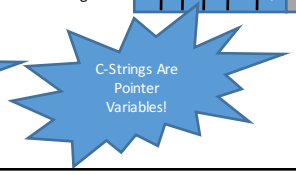




Let's Go Back To C-Strings!

```
char cstring[80];
strcpy( cstring, "Hello" );
```





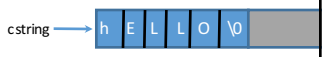
Let's Go Back To C-Strings!

```
char cstring[80];
strcpy( cstring, "Hello" );
```


```
switchCase( cstring );
```

Let's Go Back To C-Strings!

```
char cstring[80];
strcpy( cstring, "Hello" );
```




```
switchCase( cstring );
```

switchCase Function

```
void switchCase( char * data )
{



}
```

switchCase Function

```
void   switchC a se (    char   * data   )
{
     char   *  ptr  =  data;


}
```

switchCase Function

```
void   switchC a se (    char   * data   )
{
     char   *  ptr  =  data;
     while   (*ptr   !=  '\0')
     {


          ptr   =  ptr  +  1;
     }
}
```

switchCase Function

```
void   switchC a se (    char   * data   )
{
     char   *  ptr  =  data;
     while   (*ptr   !=  '\0')
     {
          //   change   the   case...
          char   letter    = *ptr;









          ptr   =  ptr  +  1;
     }
}
```

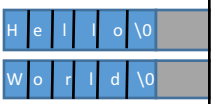
switchCase Function

```
void switchCase( char * data )
{
    char * ptr = data;
    while (*ptr != '\0')
    {
        // change the case...
        char letter = *ptr;
        int offset = 0;
        if (isupper(letter))
        {
            offset = 32;
        }
        else if (islower(letter))
        {
            offset = -32;
        }
        *ptr = *ptr + offset;
        ptr = ptr + 1;
    }
```

Let's Go Back To C-Strings!

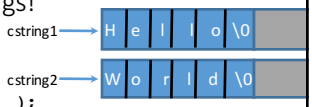
```
char cstring1[80];
char cstring2[80];
strcpy( cstring1, "Hello" );
strcpy( cstring2, "World" );
```

Let's Go Back To C-Strings!




```
char cstring1[80];
char cstring2[80];
strcpy( cstring1, "Hello" );
strcpy( cstring2, "World" );

if (!greaterThan( data1, data2 ))
{
   cout << "data1 IS NOT > data2" << endl;
}
```

Let's Go Back To C-Strings!




```
char cstring1[80];
char cstring2[80];
strcpy( cstring1, "Hello" );
strcpy( cstring2, "World" );

if (greaterThan( data2, data1 ))
{
   cout << "data2 IS > data1" << endl;
}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{


}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;



   return( result );
}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;
   char * ptr2 = cstring2;


   return( result );
}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;
   char * ptr2 = cstring2;
   // letter by letter comparison
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        ptr1 = ptr1 + 1;
        ptr2 = ptr2 + 1;
   }
   return( result );
}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstring2;
   // letter by letter comparison
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   return( result );
}
```

greaterThan Function

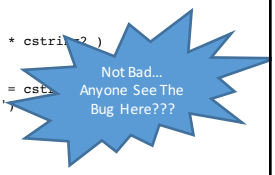
```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstring2;
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        if (*ptr1 > *ptr2) {
            result = true;
            break;
        }
        else if (*ptr1 < *ptr2) {
            result = false;
            break;
        }
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   return( result );
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstring2;
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        if (*ptr1 > *ptr2) {
            result = true;
            break;
        }
        else if (*ptr1 < *ptr2) {
            result = false;
            break;
        }
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   return( result );
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstr
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        if (*ptr1 > *ptr2) {
            result = true;
            break;
        }
        else if (*ptr1 < *ptr2) {
            result = false;
            break;
        }
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   return( result );
```



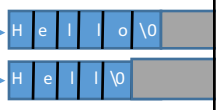
greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstring2;
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        if (*ptr1 > *ptr2) {
            result = true;
            break;
        }
        else if (*ptr1 < *ptr2) {
            result = false;
            break;
        }
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   return( result );
```

Not Bad… Anyone See The Bug Here???

What If The Strings Have Different Lengths???

Let's Go Back To C-Strings!




```
char cstring1[80];
char cstring2[80];
strcpy( cstring1, "Hello" );
strcpy( cstring2, "Hell" );

if (greaterThan( data1, data2 ))
{
   cout << "data1 IS > data2" << endl;
}
```

greaterThan Function

```
bool greaterThan( char * cstring1, char * cstring2 )
{
   bool result = false;
   char * ptr1 = cstring1;  char * ptr2 = cstring2;
   while (*ptr1 != '\0' && *ptr2 != '\0')
   {
        if (*ptr1 > *ptr2) {
            result = true;
            break;
        }
        else if (*ptr1 < *ptr2) {
            result = false;
            break;
        }
        ptr1 = ptr1 + 1;  ptr2 = ptr2 + 1;
   }
   // there might be letters left over...
```

greaterThan Function

```
// there might be letters left over...
// are there still any letters left??
    if (*ptr1 != '\0')
    {
        result = false;
    }
    else if (*ptr2 != '\0')
    {
        result = true;
    }
    return( result );
}
```

Compare Pointers To Reference Variables

- In C++, Pointers And Reference Variables Are Very Similar…

Compare Pointers To Reference Variables

- In C++, Pointers And Reference Variables Are Very Similar…

```
void foo( int & i );
void bar( int * i );
```

Compare Pointers To Reference Variables

- In C++, Pointers And Reference Variables Are Very Similar…

```
void foo( int & i );
void bar( int * i );
```

- Foo Gets Passed An Existing L-Value From Driver Code
- C++ Knows How To Convert An L-Value Into It's Address
- As In: `int x = 12;  foo( x );`
- The Parameter i Can **Never** Be `nullptr`

Compare Pointers To Reference Variables

- In C++, Pointers And Reference Variables Are Very Similar…

```
void foo( int & i );
void bar( int * i );
```

- Bar Gets Passed An Address From Driver Code.
- As In: `int x = 12;  bar( &x );`
  As In: `int*y = &x;  bar( y );`
- But Be Careful! It Might Be nullptr
- As In: `int*y = nullptr;  bar( y );`

Passing Pointers By Reference!

- Suppose We Want A Function To Change A Pointer's Value (The Arrow, Not The Box….)

Passing Pointers By Reference!

- Suppose We Want A Function To Change A Pointer's Value (The Arrow, Not The Box….)

```
int a = 12;
int b = 13;
int * ptrA = &a;
int * ptrB = &b;
```

Passing Pointers By Reference!

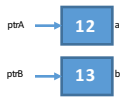
- Suppose We Want A Function To Change A Pointer's Value (The Arrow, Not The Box….)

```
int a = 12;
int b = 13;
int * ptrA = &a;
int * ptrB = &b;
```

Passing Pointers By Reference!

- Suppose We Want A Function To Change A Pointer's Value (The Arrow, Not The Box….)

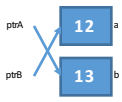
```
int a = 12;
int b = 13;
int * ptrA = &a;
int * ptrB = &b;

swapArrows( ptrA, ptrB );
```



Compare Pointers To Reference Variables

- Suppose We Want A Function To Change A Pointer's Value (The Arrow, Not The Box….)

```
int a = 12;
int b = 13;
int * ptrA = &a;
int * ptrB = &b;

swapArrows( ptrA, ptrB );
```

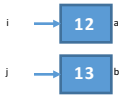


Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```

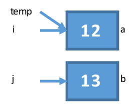
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



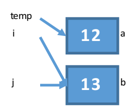
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



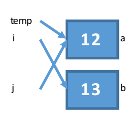
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



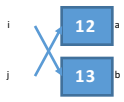
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



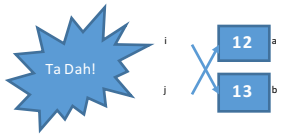
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



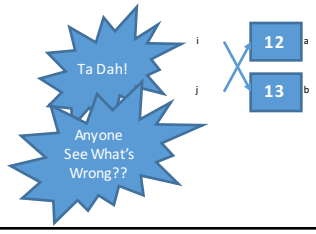
Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



Passing Pointers By Reference!

```
void swapArrows( int * i, int * j )
{
    int * temp = i;
    i = j;
    j = temp;
}
```



Passing Pointers By Reference!

```
void swapArrows(  int * & i, int * & j )
{
     int * temp = i;
     i = j;
     j = temp;
}
```