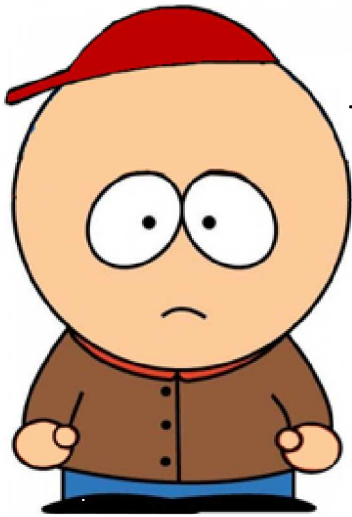
C++ Programming

Bits and Pieces

This IS a programming course!

I haven't programmed in C++ much in the past year or so. Do you think I can do OK in this course?

I just finished CS 221 and I have a 4.0 grade point average in all CS courses.



C++ Programming

Bits and Pieces

- **Data types and Operators**
- **Pointers and References**
- **Functions**
- **Strings**
- **Standard Template Library**

Some of the material in these topics may be new to you while some may be just a review. All topics cover some aspect of programming you may find useful in the programming assignments.

Bits and Pieces

Variables and Operators

C++ Data Types: Integers

char 
Size: 8 bits
Signed: -128..127
Unsigned: 0..255

bool 
Size: 8 bits
true (NOT 0) or
false (0)

short 
Size: 16 bits
Signed: -32768..32767
Unsigned: 0..65535

int 
In the first implementation of the C language int was 2 bytes.
Size: 32 bits
Signed: -2,147,483,648..2,147,483,647
Unsigned: 0..4,294,967,295

long 
Size: 32 bits
Signed: -2,147,483,648..2,147,483,647
Unsigned: 0..4,294,967,295

long long 
Size: 64 bits
Signed: -9,223,372,036,854,775,808.. 9,223,372,036,854,775,807
Unsigned: 0..18,446,744,073,709,551,615 (that's over 10^{18})

C++ Data Types: Floating point

float 
Size: 4 bytes (32 bits)
$\pm(10^{-38}..10^{38})$ with 7 digits of precision
Digits-23 bits, Exponent-8 bits, Sign-1 bit

double 
Size: 64 bits
$\pm(10^{-307}..10^{308})$ with 15 digits of precision
Digits-47 bits, Exponent-16 bits, Sign-1 bit

long double
Size: 64 bits (8 bytes) 
80 bits (10 bytes) 
96 bits (12 bytes) 
128 bits (16 bytes) 
Platform dependent. On a PC it is 64 bits (same as a double) with 15 digits of precision). On a Macintosh it is 128 bits with 32 digits of precision.

Bits and Pieces

Enumerated Data Types

Suppose you wanted something like this...

```
#define NORTH_WIND      0
#define SOUTH_WIND      1
#define EAST_WIND       2
#define WEST_WIND       3
#define NO_WIND         4

int wind_direction = NO_WIND;
```

But, what is there to prevent someone from doing this?

```
wind_direction = 64;
```

Declaring *enum* data types

```
                                0          1          2          3          4
enum WindDir {NORTH_WIND, SOUTH_WIND, EAST_WIND, WEST_WIND, NO_WIND}
                0     1     2     3     4     5     10          11
enum Number {ZERO, ONE, TWO, THREE, FOUR, FIVE, TEN = 10, ELEVEN}
                1     2       3       4      5     6     7       8      9     10
enum Cards {ACE=1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
                    11      12      13
            JACK, QUEEN, KING}
```

- First item in an enum is assigned the value 0 unless set to another value.
- Each succeeding item is incremented by 1.
- Any item can be assigned a specific value if it is greater than the previous item and less than the next item.
- The values of all items must be ordered from least to most.

Warning: If both of these were defined in the same file you would get name comflicts on TWO, THREE, FOUR, etc.

Bits and Pieces

Enumerated Data Types

Defining variables of the *enum* types

```
WindDir dir = NORTH_WIND;
Number num = ZERO;
Cards card1 = ACE;
Cards card2 = JACK;
```

Using *enum* variables

```
if(dir == NORTH_WIND)
    cout << "Brrrrr. It's cold!";
else if(dir == SOUTH_WIND)
    cout << "Head for the beach!";
else if(dir == EAST_WIND)
    cout << "Hoist the mainsail!";
else if(dir == WEST_WIND)
    cout << "It's a homebound wind!";
```

```
switch(num)
{
        case ZERO :     // do something
                break;
        case ONE  :     // do something
                break;
        case TWO  :     // do something
                break;
        case THREE :    // do something
                break;
        case FOUR :     // do something
                break;
        // And so on...
}
```

```
if((card1 == ACE) && ((card2 == TEN) || (card2 == JACK) || (card2 == QUEEN) || (card2 == KING)))
        cout << "Blackjack!!!" << endl;
if((card1 == ACE) && (card2 >= TEN) && (card2 <= KING))
        cout << "I said, Blackjack!!!" << endl << endl;
```

Bits and Pieces

Type Casting

Old Style (C and C++)

```
double X = 3.14159265358979;    // Note 15 digits of precision
float f = (float)X;             // Lose precision. f equals 3.141592
int i = (int)X;                 // Lose precision. i equals 3
```

New Style (Four types of cast)

static_cast

Syntax: static_cast<dataTypeName>(expression)

reinterpret_cast

Syntax: reinterpret_cast<dataTypeName>(expression)

const_cast

Syntax: const_cast<dataTypeName>(expression)

dynamic_cast

Syntax: dynamic_cast<dataTypeName>(expression)

Bits and Pieces

Type Casting : static_cast

Cast	Results
static_cast<int>(7.9)	7 - ints drop the fraction, no rounding
static_cast<int>(3.3)	3
static_cast<double>(25)	25.0
static_cast<double>(15 / 2)	7.0 - perform int division first
static_cast<double>(15) / 2	7.5 - implicit cast of 2 to 2.0 then divides into 15.0
static_cast<int>(7.9 + 6.7)	14 - adds to get 14.6 then casts
static_cast<int>(7.9) + static_cast<int>(6.7)	13 - casts 7.9 to 7 then 6.7 to 6 before adding
static_cast<int>('A')	65 - ASCII value of 'A'
static_cast<char>(65)	'A'

Bits and Pieces

Type Casting : reinterpret_cast

Cast From	Cast To	Example
A pointer of any type	An integer large enough to hold it, i.e. an int or long data type.	int x; int *iptr = &x; long xAddr; xAddr = reinterpret_cast<long>iptr
A short, int, or long	A pointer of any type	int x = 0; long *ptr; ptr = reinterpret_cast<long *>x
A pointer to a function *This may not mean anything now. But, it will....;-)*	A pointer to a function of a different type.	char str[] = "Just a test"; typedef long (*FUNC)(const char*); FUNC fptr1, fptr2; fptr1 = strlen; fptr2=reinterpret_cast<FUNC>strcat; fptr2(str); // Crash and burn
A pointer to an instance of a class or structure	A pointer to an instance of a different type of class or structure	simple *s = new simple(); complex *c; c = reinterpret_cast<complex *>s; c->d = 2.5; // Crash and burn

Bits and Pieces

Type Casting : const_cast

Simple function

Simple function with *const* arg.

Two *const* variables

```
#include <iostream>
using namespace std;

void f(int *p)
{
  cout << *p << endl;
}

void g(const int *p)
{
  cout << *p << end1;
}

int main(void)
{
  const int a = 10;
  const int* b = &a;
```

Example 1

```
// Function f() expects int*, not const int*
//   so we cast off the const of b
int *c = const_cast<int *>(b);
f(c);  // This call works, f(b) would not
```

Example 2

```
// L value is const
//  *b = 20;   // can't do this, a is const
```

Example 3

```
// Undefined behavior
//  *c = 30;    // can't do this, a is const
```

Example 4

```
int a1;
int *b1 = &a1;

g(b);   // OK b is const
g(const_cast<int *>(&a1));  // OK,
                            // makes a1 const.
g(const_cast<int *>(b1));   // Also OK,
                            // makes b1 const.

 return 0;
} // end main()
```

Bits and Pieces

Type Casting : dynamic_cast

```
cShape *s = new cRectangle();
cRectangle *r;
r = dynamic_cast<cRectangle *>(s);
```

Bits and Pieces

Operators

All of these you should know.

Math

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Mod-division
++	Increment by 1
--	Decrement by 1

Operator Equals
+=, -=, *=, /= , %=

Logical

Given: bool A = true, bool B = false, int X=5, int Y=10

&&	Logical AND, if (A && B) is false
||	Logical OR, if (A || B) is true
!	Logical NOT, if (!A) is false
==	Logical equals, if (X == Y) is false
<	Less than, if (X < Y) is true
>	Greater than, if (X > Y) is false
<=	Less than or equal to, if (X <= Y) is true
>=	Greater than or equal to, if (X >= Y) is false
!=	Not equal, if (X != Y) is true

Bitwise

Given: int X=10 (hex 0x0A, binary 0000 1010)
int Y= 3 (hex 0x03, binary 0000 0011)

&	Bitwise AND	(X & Y) = 2 (0x02, 1010 & 0011 = 0010)
|	Bitwise OR	(X | Y) = 11 (0x0B, 1010 & 0011 = 1011)
~	Bitwise NOT	(~X) = -11 (1111 0101) (~Y) = -4 (1111 1100)
^	Bitwise XOR	(X ^ Y) = 9 (1010 ^ 0011 = 1001)
<<	Bit shift left	(X << Y) = 80 (0x50 1010 << 0011 = 0101 0000)
>>	Bit shift right	(X >> Y) = 1 (1010 >> 0011 = 0001)

*Note: Most operating systems use twos-complement to encode integers in binary

All these also work with the = sign like the math operators, e.g. X&=Y

Other

( )	Parentheses
[ ]	Brackets
.	Member selection with object name
->	Member selection with pointer

You may not have thought of these as operators.

Bits and Pieces

Operator Overloading

- Any of the operators can be overloaded.
- Many already have been.

Example of overloading in a linked list module:

```
class ListNode
{
    private:
        int key;
        // Other vars defined here
    public:
        ListNode();
        ~ListNode();
        int getKey();
        // Other functions defined here
        bool operator<(ListNode n);
}
```

Note: No *

```
class OrderedList
{
    private:
        ListNode *head;
        // Other vars defined here
    public:
        OrderedList();
        ~OrderedList();
        bool insert(ListNode *n);
        // Other functions defined here
        bool operator+=(ListNode *n);
}
```

Note: Has *

Bits and Pieces

Operator Overloading

Example of overloading in a linked list module:

```
// Overload < operator for comparing
//    this node's key with another node
bool ListNode::operator<(ListNode n)
{
    return(key < n.getKey());
}
```

Note: No *

```
// Overload += operator to allow using it
//    as an insert operator
bool OrderedList::operator+=(ListNode *n)
{
    return insert(n);
}
```

Note: Has *

```
// Code sample
// Assume *n1 and *n2 point to 2 ListNode
//   objects
if(*n1  < *n2)
{
   // insert n1 before n2 in the list
}
```

Note: Both have *

```
// Code sample
// Assume *n1 is a new ListNode object to be
//  inserted into the OrderedList referenced by
//  the pointer *OLst;
n1 = new ListNode();
// Add data to n1
*OLst += n1;  // Insert n1 into the list
```

Note: No *

Note: Has *

Bits and Pieces

Pointers and References

Review of Pointers

Pointer – A variable that can hold the memory address of another variable. Pointers are type specific.

```
int       *m_ipPtr;      // Pointer to an int
long      *m_lpPtr;      // Pointer to a long
float     *m_fpPtr;      // Pointer to a float
double    *m_dpPtr;      // Pointer to a double
char      *m_cpPtr;      // Pointer to a char
simple    *m_opSim;      // Pointer to an instance of a struct simple
MyClass   *m_opMC;       // Pointer to an instance of class MyClass
```

Note the naming convention used.

Bits and Pieces

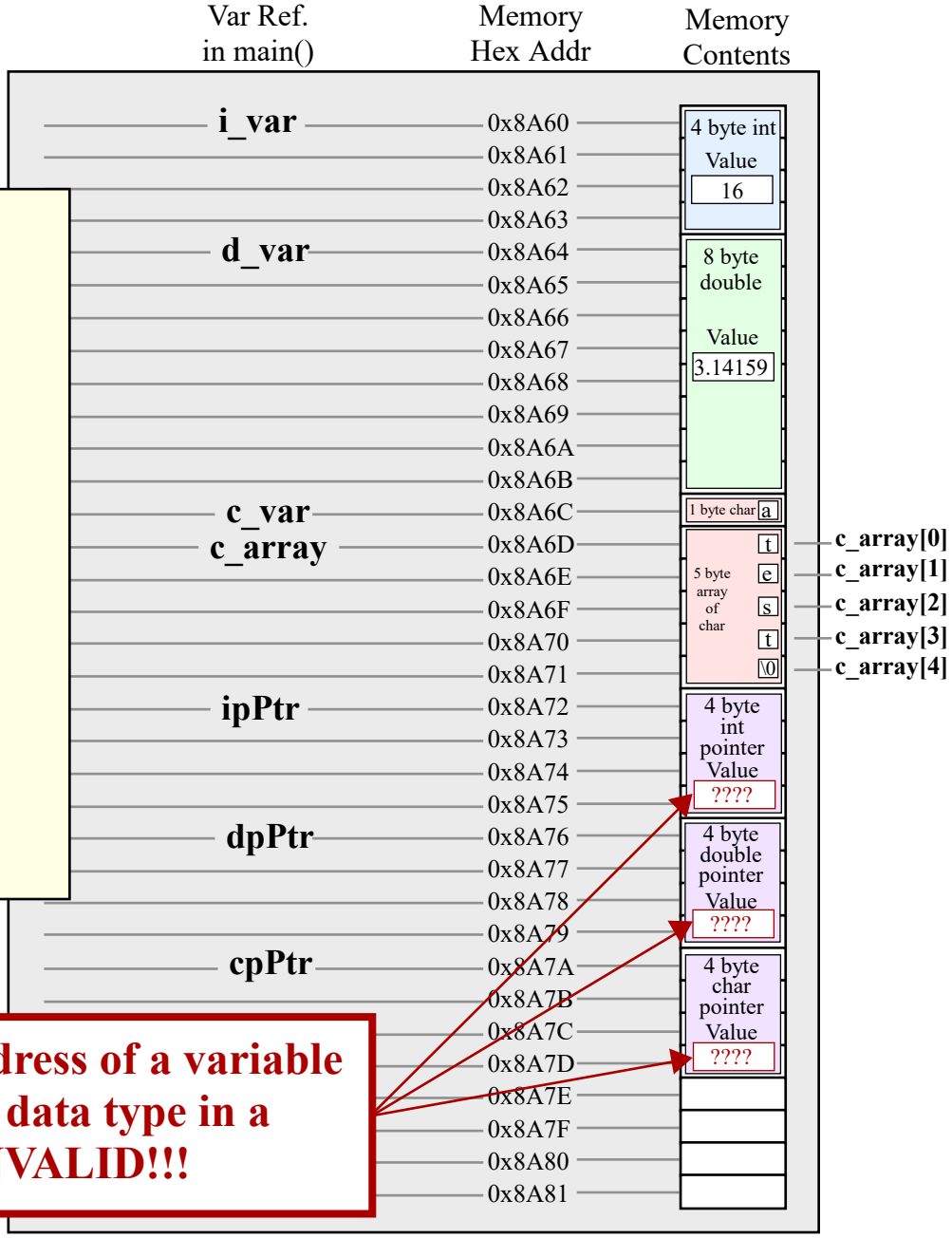
Review of Pointers

What's happening in memory?

Execute these commands then take a look at memory.

```
// Create 3 variables and a char array
int         i_var = 16;
double      d_var = 3.14159;
char        c_var = 'a';
char        c_array[5] = "test";

// Create 3 pointers
int         *ipPtr;     // Pointer to int
double      *dpPtr;     // Pointer to double
char        *cpPtr;     // Pointer to char
```

*Some programmers like to use prefixes such as **ip**, **dp**, **cp**, etc. in the names of pointer variables to make them easier to identify in code.*

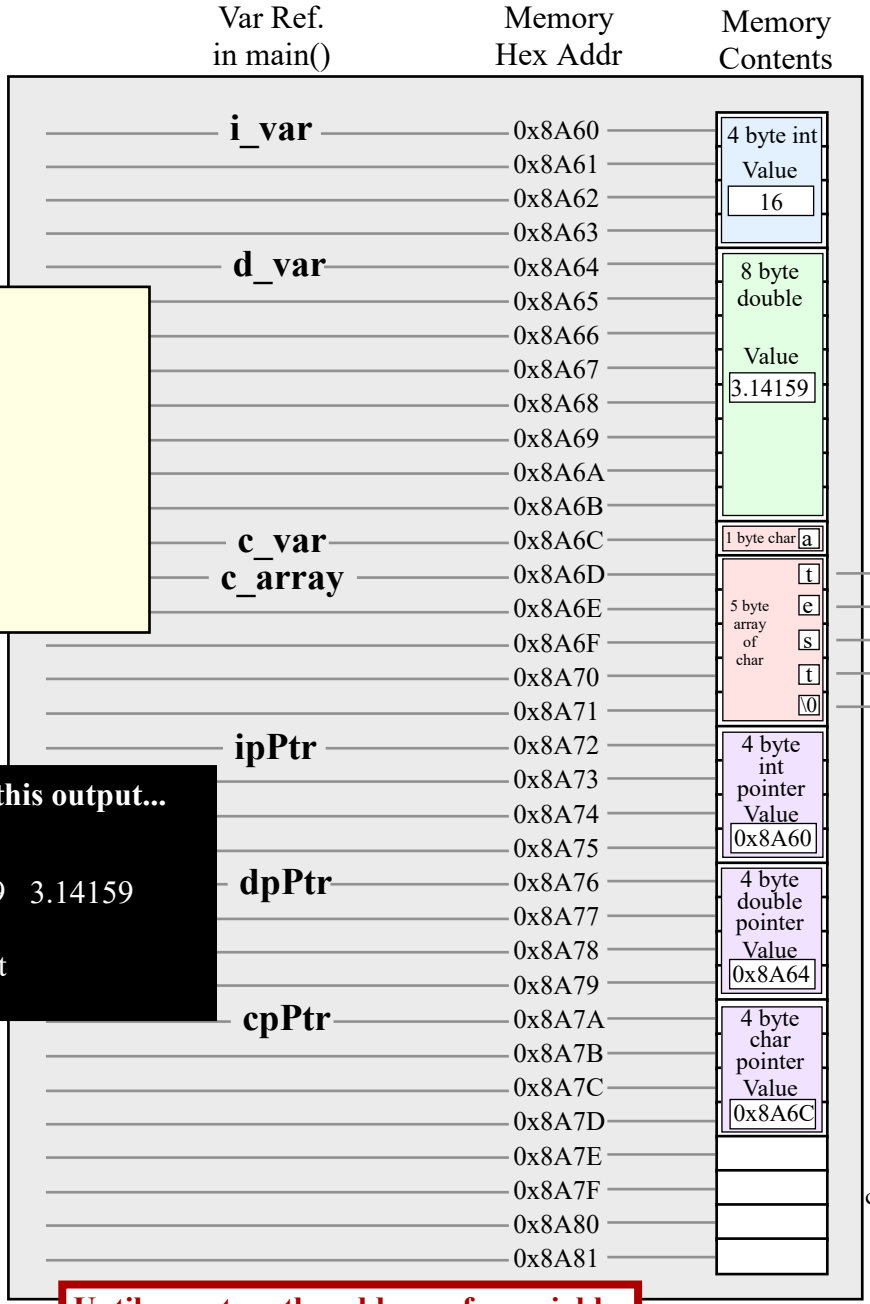


Until you store the address of a variable of the appropriate data type in a pointer it is INVALID!!!

You can initialize the contents of a char array when it is created. After that you must use strcpy(c_array, "test"); Remember when 25% of the CS 221 class tried to do this... c_array = "test"; This is still a Syntax Error.

Review of Pointers

What's happening in memory?



Now, execute these commands then take a look at memory.

```
ipPtr = &i_var;
dpPtr = &d_var;
cpPtr = &c_var;
```

Given this code...

```
cout << i_var << " " << *ipPtr << endl;
cout << d_var << " " << *dpPtr << endl;
cout << c_var << " " << *cpPtr << endl;
cpPtr = c_array;
cout << c_array << " " << cpPtr << endl;
```

You get this output...

```
16  16
3.14159  3.14159

test  test
```

Were you paying attention to this on the previous slide?

Until you store the address of a variable of the appropriate data type in a pointer it is INVALID!!!

Review of Pointers

Using pointers to change the values in variables they point to.

If this code has been executed...

```
ipPtr = &i_var;
dpPtr = &d_var;
cpPtr = &c_var;
```

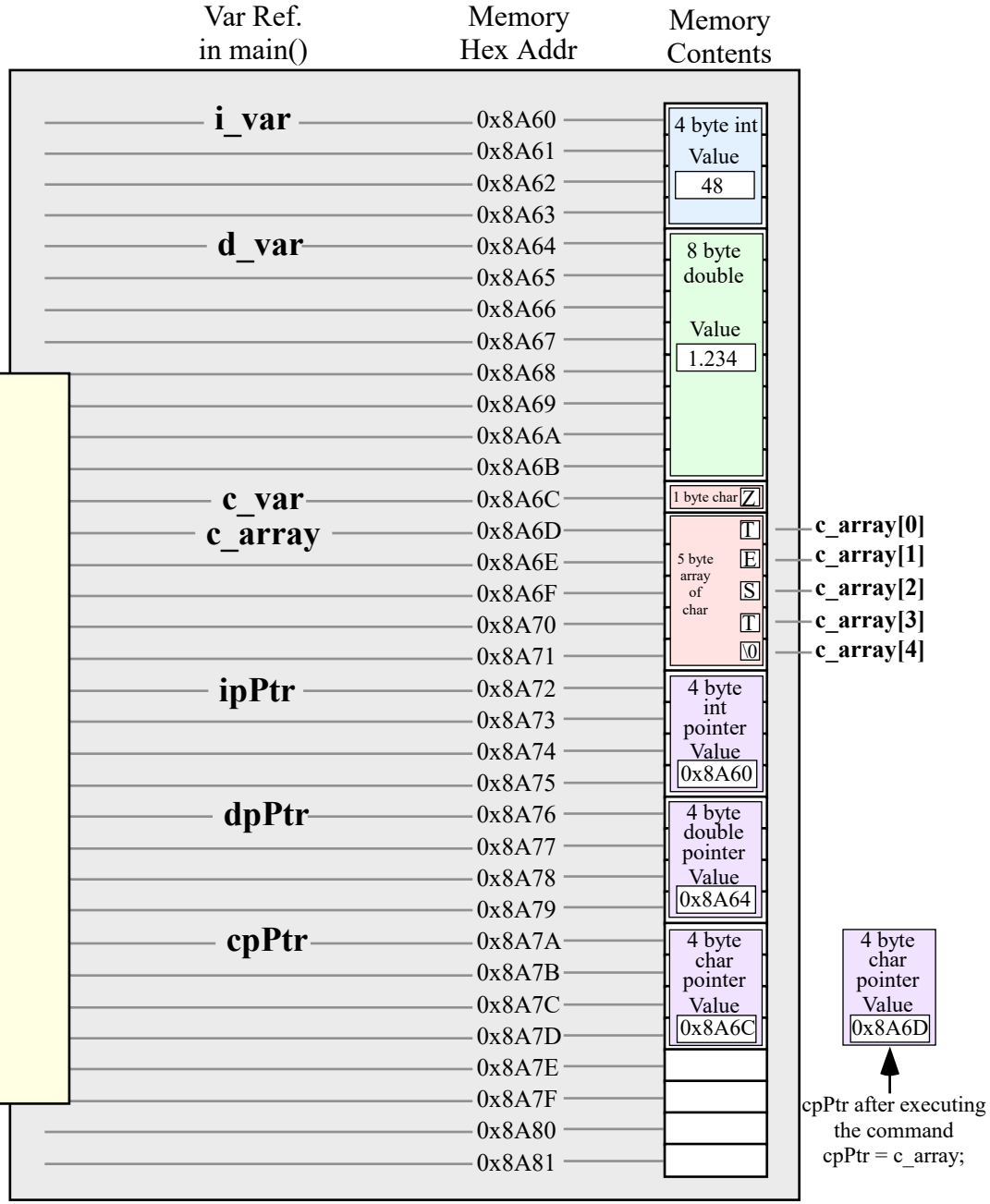
And, then you execute these commands...

```
*ipPtr = 48;
*dpPtr = 1.234;
*cpPtr = 'Z';
```

Take a look at memory...

And, then execute these commands...

```
cpPtr = c_array;
strcpy(cpPtr, "TEST");
```



Bits and Pieces

Review of Pointers

Watch out for this...

Code Executed in main()

```
int          i_var = 16;
double       d_var = 3.14159;
char         c_var = 'a';
int          *i_ptr;
double       *d_ptr;
char         *c_ptr;
// Note: the pointers have not been set to point to anything
MyFunc(&i_var, &d_var, &c_var);  // You're OK here
MyFunc(i_ptr, d_ptr, c_ptr);     // Crash and burn here
```

Just where in memory are you storing these values?

```
void MyFunc(int *arg1, double *arg2, char *arg3)
{
   // Input values
   cout << "Enter an int, double, and char.");
   cin >> *arg1 >> *arg2 >> *arg3;
   // Print  values, unless you just crashed
   cout << *arg1 << endl;
   cout << *arg2 << endl;
   cout << *arg3 << endl;
}
```

Bits and Pieces

Review of Pointers

...and this

A very common screw up of this type!

```
// Screw up #1
char *name;
GetFileName(name);
```

```
// Screw up #2
char name;
GetFileName(&name);
```

Just where in memory are you placing this copy?

```
// Now this will work
char name[128];
bool nameOK;
nameOK = GetFileName(name);
```

```
bool GetFileName(char *name)
{
    char fileName[64];
    cout << "Enter the data file name";
    cin.getline(fileName, 63, '\n');
    // Check to make sure the name is
    //    valid. If not return false.
    strcpy(name, fileName); // Copy file name
    return true; // Got a valid file name
}
```

Bits and Pieces

Pointers to Structures and Classes

These pointers haven't been set pointing to anything yet!

```
simple      m_oSim1;        // Static instance of a struct simple
simple      *m_opSim;       // Pointer to an instance of a struct simple
MyClass     m_pMC1;         // Static instance of class MyClass
MyClass     *m_opMC;        // Pointer to an instance of class MyClass

m_opSim = new simple();     // Create dynamic instances
m_opMC = new MyClass();     // using the new operator...*

// Store values in the structs...
m_oSim1.x = 32;             // Using "dot" notation
m_oSim1.ch = 'a';
m_opSim->x = 16;            // Using "pointer" notation
m_opSim->ch = 'z';

// Store values in the classes...
m_oMC1.setX(16);            // Using "dot" notation
m_oMC1.setCh('A');
m_opMC->setX(48);           // Using "pointer" notation
m_opMC->setCh('Z');
```

These commands do NOT create dynamic instances of the objects

```
m_opSim = &m_oSim1;
m_opMC = &m_pMC1;
```

In case you don't remember these.

```
struct simple
{
    int x;
    char ch;
};
```

```
class MyClass
{
    private
        int x;
        char ch;
    public:
        MyClass();
        ~MyClass();
        int getX();
        void setX(int x);
        char getCh();
        void setCh(char c);
};
```

Bits and Pieces

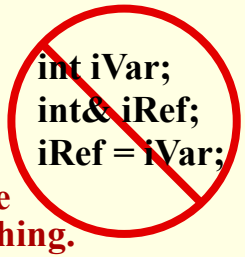
Reference Variables

Reference – A variable that is an alias for another variable.

Rules for Reference Variables

- **Reference variables must be initialized when created.**

```
int iVar;              // Create an int variable
int& iRef = iVar;      // Create an int reference and set it to iVar
```



You cannot create a reference then set it to reference something.

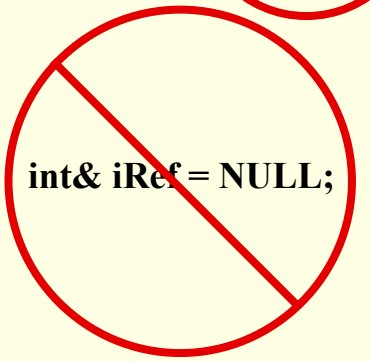
- **Once set a reference variable cannot be changed.**

```
int iVar, iVar2;       // Create 2 int variables
int& iRef = iVar;      // Create an int reference and set it to iVar
```

You cannot change a reference once set.

- **Cannot have a NULL reference variable.**

This won't work at all.



We will look at these again with functions.

Bits and Pieces

Functions

- The Basics
- Function Prototyping
- Function Overloading
- Function Calling
- Default Arguments
- Variable Argument Lists
- Function Pointers
- Virtual Functions
- Friend Functions

The Basics

```
returnType className::functionName(argument list)
{
      function body
}
```

Bits and Pieces

Function Prototyping

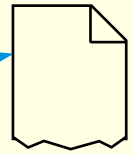
The compiler has to know the proper syntax for calling a function, before it can compile any code making calls to the function.

```
// Prototyping in procedural
//    programs

#include <iostream>
using namespace std;

int Function1(int, double);

void main()
{
    int x = 3;
    double d = 2.5;

    Function1(x, d); // Call function
}

int Function1(int a, double b)
{
    cout << "x = " << a << "  d = " << b;
}
```

Declarations

Definitions

```
// Prototyping in classes

#ifndef ACLASS_H
#define ACLASS_H

class AClass
{
private:
      int x;
      int y;
public:
    AClass();
    ~AClass();
    void setX(int val);
    int getX();
    void setY(int val);
    int getY();
};
```



AClass.cpp

Bits and Pieces

Function Overloading

Same named functions but different argument lists.

Example:

```
bool MyList::insert(Node * newNode);
bool MyList::insert(list of args giving data to insert into a new Node);
```

But, you can't do this:

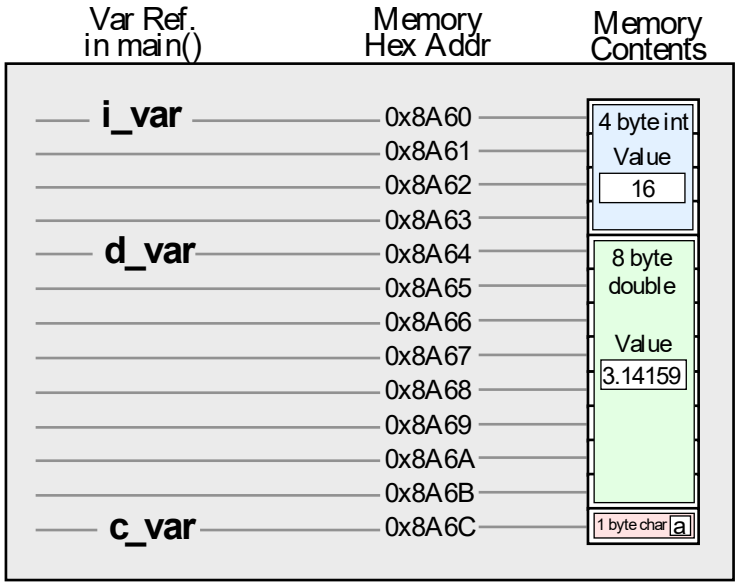
```
int MyClass::function1(int x);
double MyClass::function1(int x);

mc->function1(24);   // Which function1 is being called?
```

Bits and Pieces

Calling Functions: Call-By-Value

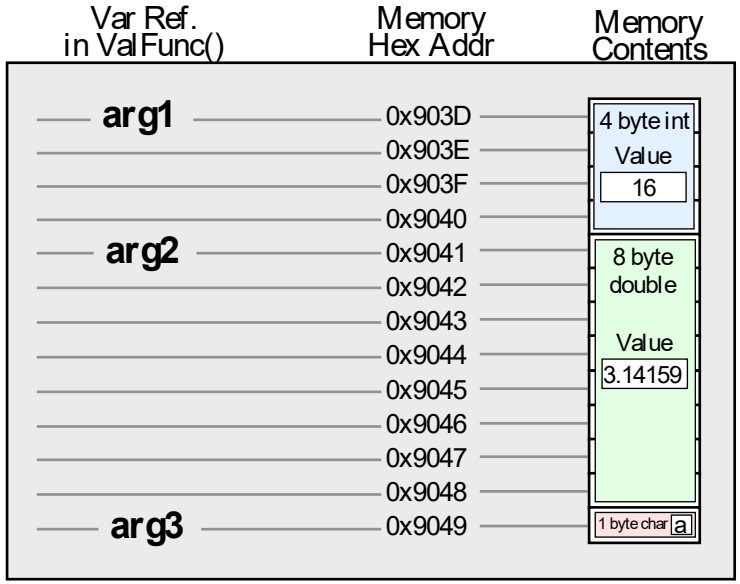
Code Executed in main()

```
int         i_var;
double      d_var;
char        c_var;

i_var = 16;
d_var = 3.14159;
c_var = 'a';

ValFunc(i_var, d_var, c_var);
```



…somewhere else in memory

```
void ValFunc(int arg1, double arg2, char arg3)
{
    // Function code is here.  References to
    //   arg1, arg2, and arg3 are to the new
    //   variables created by the call to the
    //   function.  Only the values from i_var
    //   d_var, and c_var were passed in.
    cout << arg1 << endl;  // print  value 16
    cout << arg2 << endl;  // print  value 3.14159
    cout << arg3 << endl;  // print  value 'a'
}
```



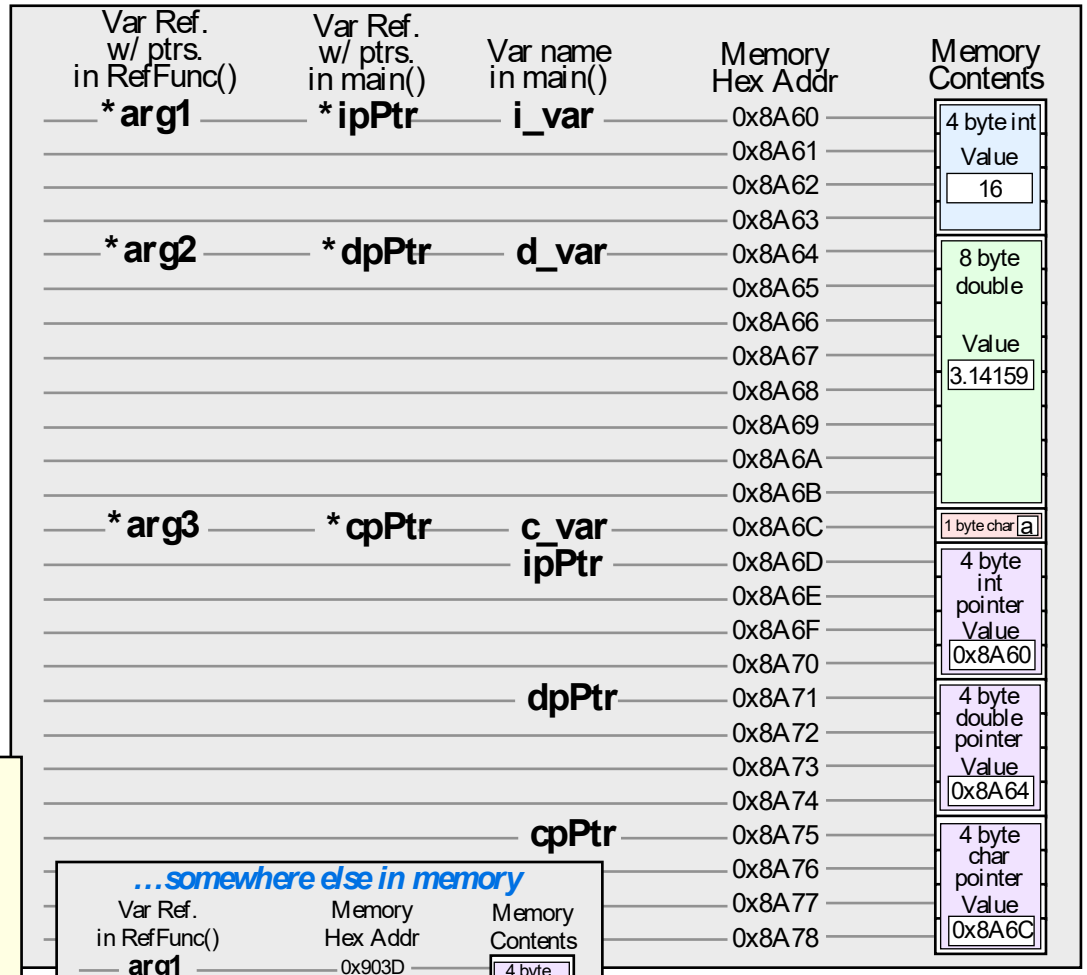
Call By Reference (using pointers in C or C++)

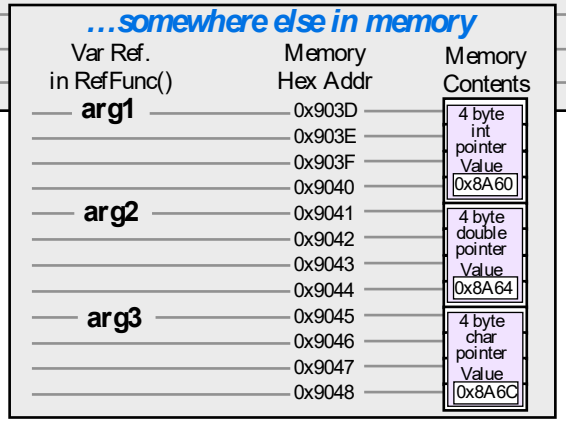
Code Executed in main()

```
int       i_var = 16;
double    d_var = 3.14159;
char      c_var = 'a';
int       *ipPtr;
double    *dpPtr;
char      *cpPtr;

ipPtr = &i_var;
dpPtr = &d_var;
cpPtr = &c_var;

RefFunc(&i_var, &d_var, &c_var);
RefFunc(ipPtr, dpPtr, cpPtr);
```

"Reference" function using pointers

```
void RefFunc(int *arg1, double *arg2, char *arg3)
{
   // Function code is here.  References using
   //   *arg1, *arg2, and *arg3 are to the
   //   variables i_var, d_var, and c_var
   //   using the address which were passed in
   //   "by value."
    cout << *arg1 << endl;  // print  i_var
    cout << *arg2 << endl;  // print  d_var
    cout << *arg3 << endl;  // print  c_var
}
```



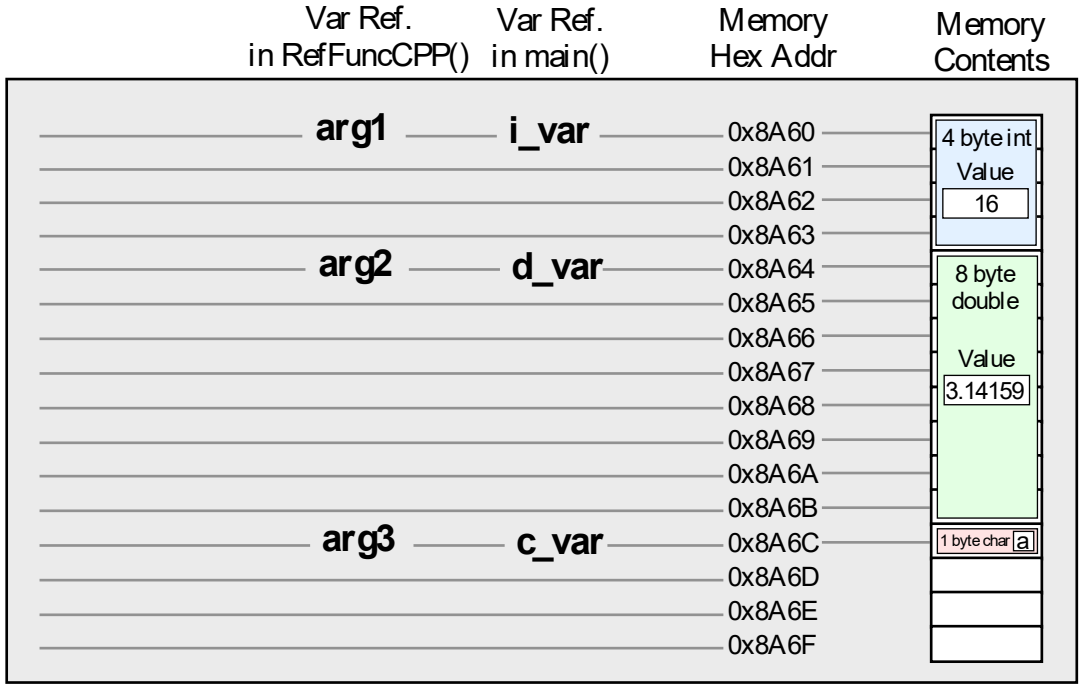


Bits and Pieces

Call By Reference (using references in C++)

Code Executed in main()

```
int          i_var = 16;
double       d_var = 3.14159;
char         c_var = 'a';

RefFuncCPP(i_var, d_var, c_var);
```



…somewhere else in memory

```
void RefFuncCPP(int& arg1, double& arg2, char& arg3)
{
   // Function code is here.  Arguments
   //   arg1, arg2, and arg3 are aliases to
   //   variables i_var, d_var, and c_var.
   cout << arg1 << endl;  // print  i_var
   cout << arg2 << endl;  // print  d_var
   cout << arg3 << endl;  // print  c_var
}
```

Bits and Pieces

Call By Reference - Pointer References

...in main()...

```
int valArray[5] = {5, 10, 25, 50, 100};
int *iptr = valArray;
cout << "*iptr = " << *iptr << endl;

incPointerWithHandle(&iptr);
cout << "*iptr = " << *iptr << endl;

incPointerWithReference(iptr);
cout << "*iptr = " << *iptr << endl;
```

Output

```
*iptr = 5
*iptr = 10
*iptr = 25
```

Each of these functions increments the pointer passed as an argument so that it points to the next int value in the valArray

Increment Using A Pointer to a Pointer (Handle)

```
void incPointerWithHandle(int **h)
{
        (*h)++;
}
```

*Note the parentheses which are required because of the binding of **

Increment Using A Reference to a Pointer

```
void incPointerWithReference(int *&r)
{
        r++;
}
```

Arg r is a reference to an int pointer

Bits and Pieces

Default Arguments

A simple function

```
void someFunction(int x, double d, char c, long l)
{     // Function body...     }
```

The prototype for a simple function with default values for arguments

```
void someFunction(int x = 0, double d = 1.0, char c = 'a', long l = 1);
```

Note: the default values go in the prototype only, not in the function definition.

Various legal calls to the function

```
someFunction(1, 2.0, 'z', 32);      // normal calling format
someFunction(1, 2.0, 'z');          // 3 args (arg l takes default value)
someFunction(1, 2.0);               // 2 args (args c and l take default values)
someFunction(1);                    // 1 arg (args d, c, l take default values)
someFunction();                     // 0 args (all take default values)
```

And one illegal call to the function

```
someFunction(1, 2.0, 32);     // left out a char argument
```

Bits and Pieces

Default Arguments in Constructors

A constructor prototyped with default args in the DemoClass.h file

```
DemoClass(int x = 0, double d = 1.5, char c = 'a', long l = 1);
```

A constructor function which assigns default values for certain member variables

You can use either format.

```
SomeClass::SomeClass()
{
    m_iVar1 = 0;              // int
    m_dVar2 = 0.0;            // double
    m_dPoint.x = 0;           // Point struct
    m_dPoint.y = 0;
    m_sStr = "Just a test";   // String object
                              //  not K&R array
    // Rest of the body of the function is here
}
```

```
SomeClass::SomeClass() :
    m_iVar1(0),
    m_dVar2(0.0),
    m_dPoint(0, 0),
    m_sStr("Just a test"),
{
    // body of the function is here
}
```

Do NOT try to set a default value for a member variable in the header file even if VS 2015 will let you do so.

Bits and Pieces

Variable Argument Lists

this

int function()

in C++ equals int function(void)

in C equals int function(...)

but, this

int function(...)

in either language equals...

A function with variable argument lists

Stay tuned for the next exciting slide...

Bits and Pieces

Variable Argument Lists

```
// Variable argument lists

#include <iostream>
#include <cstdarg>
using namespace std;

double average(int num, ...);

void main()
{
   cout << average(3, 12.2, 22.3, 4.5) << endl;
   cout << average(5, 3.3, 2.2, 1.1, 5.5, 3.3) << endl;

}
```

Got to have this.

Got to tell it how many arguments there will be.

```
//-----------------------------------------
// Variable arg list function
//-----------------------------------------
double average(int num, ...)
{
    // Make a place to store the list of arguments
   va_list arguments;
   double sum = 0;

   // Initializing arguments to store all values after num
   va_start (arguments, num);
   for (int i = 0; i < num; i++ )
   {
      // Add the next value in argument list to sum.
      sum += va_arg(arguments, double);
   }
   va_end(arguments);          // Clean up the list

   return (sum/num);
}
```

The classic variable argument list functions

Prototypes

```
int scanf(const char *format, ...);
int printf(const char *forma, ...);
```

Examples in Use

```
scanf("%d %f %s", &x, &y, str);
printf("x=%d, y=%.2f, str=%s", x, y, str);
```

Assumes: int x; double y; char str[64];

Hold on to your...!!!

Bits and Pieces

Function Pointers

Pick a function, any function...

```
// Some function somewhere

void myFunction(int x)
{
        cout << “x = “ << x << end1;
}
```

...create a pointer to that function...

```
void (*funcPtr)(int);          // Declare a function pointer

funcPtr = myFunction;  // Set the pointer to myFunction

funcPtr(255);                  // Call the function using the pointer
```

You probably want to pay attention because the odds are approaching 100% that something like this will be on the next test.

Bits and Pieces

Function Pointers

```
// Another function
double myFunction(int x, double y)
{
    cout << “x = “ << x << end1;
    cout << “y = “ << y << end1;
}
```

Let’s take it one step at a time...

1. **Define a pointer and put it in parentheses**
   (*funcPtr)

2. **Add the function arguments in parentheses**
   (*funcPtr)(int, double)

3. **Add the function return type**
   double (*funcPtr)(int, double)

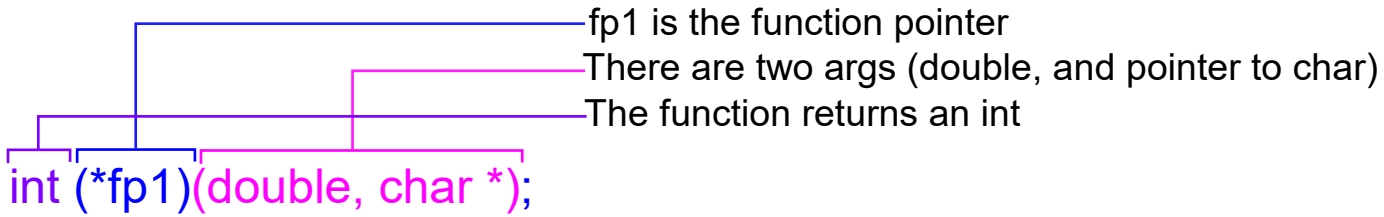
4. **Use the pointer**

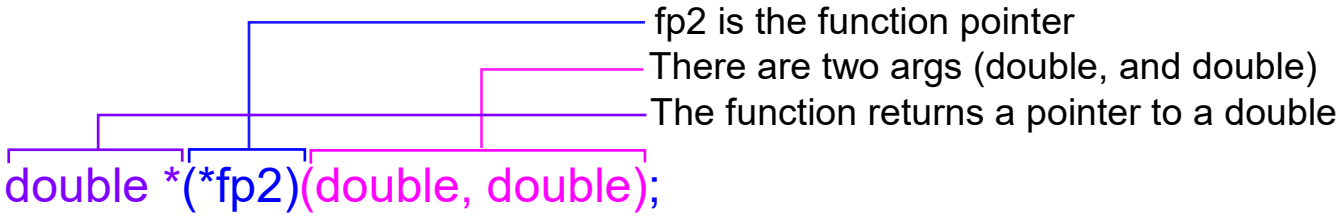
```
funcPtr = myFunction;  // Do NOT use parentheses
funcPtr(8, 16.4);  // Call the function using the pointer
```

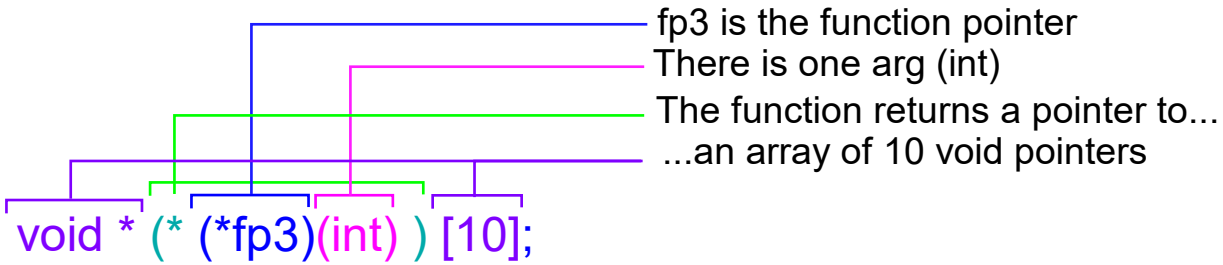
Bits and Pieces

Function Pointers

Try to wrap your mind around these...







Bits and Pieces

Function Pointers

And, how about these...

```
float (* (*fp4)(int, int, float)) (int);
```

- fp4 is the function pointer
- There are three args (int, int, and float)
- The function returns a pointer to a function...
  - ... taking an int arg and...
  - ... returning a float

```
typedef double (*(*(*fp3)())[10])();
fp3  a;  // Declare a pointer of type fp3
```

- Typedef ...
- ...fp3 as a pointer type to a function...
- ...taking no arguments
- The function returns a pointer to an..
- ...array of 10 ...
- ...pointers to functions ...
- ...each taking no arguments and ...
- ...returning double

```
/* Declaration of 5 arrays of pointers to functions returning void and taking one arg, (Widg WidgArg) */
void (*(guiMapFunction[WIDG_TYPE_COUNT]))(Widg WidgArg),
     (*(guiUnmapFunction[WIDG_TYPE_COUNT]))(Widg WidgArg),
     (*(guiDestroyFunction[WIDG_TYPE_COUNT]))(Widg WidgArg),
     (*(guiTerminateFunction[WIDG_TYPE_COUNT]))(void),
     (*(guiRefreshFunction[WIDG_TYPE_COUNT]))(Widg WidgArg);
```

- Function arg is struct Widg
- Array size
- Array names
- All functions return void

Actual code from a program developed in the early 90s. The instructor was on the team developing this GUI building application.

Bits and Pieces

Virtual Functions - What are they?

```
//-------------------------------------------------------------
// File: CShape.h
// Purpose: Demonstrate virtual functions
//-------------------------------------------------------------
#ifndef CSHAPE_H
#define CSHAPE_H
// Include headers needed to define
//   Pattern, RGBColor, Rect, etc.

class CShape
{
   protected:
      Pattern          penPat;
      Pattern          fillPat;
      RGBColor penColor;
      RGBColor fillColor;
      int          xPos;
      int          yPos;
      Rect                enclosingRect

   public:
      CShape( void );            // Initialize the object
      ~CShape( void );           // Destroy the object
      virtual void Draw();       // Subclasses will define this function
      void SetEnclosingRect(int x1, int y1, int x2, int y2);
      void SetPenPat(Pattern pPat);
      void SetFillPat(Pattern fPat);
      void SetPenColor(RGBColor pColor);
      void SetFillColor(RGBColor fColor);
};
#endif
```

```
//-------------------------------------------
// File: CRectangle.h
// Purpose: Subclass of  CShape
//-------------------------------------------
#ifndef CRECTANGLE_H
#define CRECTANGLE_H

#include CShape.h

class CRectangle:CShape
{
   // Member functi
   public:
      CRectangle( vo
      ~CRectangle( v
      void Draw();
};
#endif
```

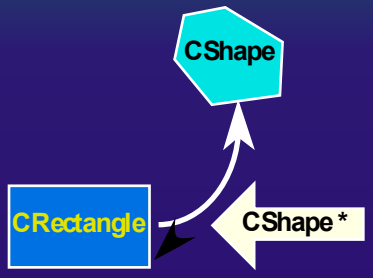
```
//-------------------------------------------
// File: COval.h
// Purpose: Subclass of  CShape
//-------------------------------------------
#ifndef COVAL_H
#define COVAL_H

#include CShape.h

class COval:CShape
{
   // Member functions
   public:
      COval( void );
      COval( void );
      void Draw();
};
#endif
```

Bits and Pieces

Virtual Functions - Why are they important?

- Early / Static / Compile time binding
  vs
  Late / Dynamic / Run time binding

- Upcasting



Bits and Pieces

Friend Functions

Friend Function - A function, not part of a class, but which has been given access to all its' private member variables and functions.

Will you be my friend?

Yes, but only if I can get to your private parts!!!

Please notice that the word "friend" does not appear anywhere here!!

```
class time_class
{
        private:
                long m_lSecs;
                friend char *presentTime(time_class tz);
        public:
                time_class(char *t);
                ~time_class();
}
```

Here is the class.h file declaring a function to be a friend.

```
char *presentTime(time_class& tz)
{
        char *ctbuf;
        ctbuf = new char[40];
        long seconds_total;

        // Next line accesses a private
        //  member variable
        seconds_total = tz. m_lSecs;
        ltoa(seconds_total, ctbuf, 10);
        return ctbuf;
}
```

Here is the function which can access the private members of an instance of the class. It is NOT A MEMBER OF THE CLASS.

Bits and Pieces

Reference Variables and Functions

...in main()...

```
// int &refA;
// refA = refFunc1();
```

1 Can't do this because refA must be initialized when it is created.

```
int &refA = refFunc1();
cout << "Got back " << refA<<
     " from call to refFunc1()\n";
```

prints 0, the value in static int x.

2 OK to do this because the variable refA references x from reFunc1 which is static and thus created outside the scope of reFunc1.

```
int& refFunc1()
{
    int q;
    // return q;
    static int x = 0;
    return x;
}
```

3 Can't do this because q is a local variable.

4 x is *static* and therefore outside the scope of the function so a reference to it can be returned. This gives access outside of a function to a static variable inside the function.

```
int X = 5;
cout << "Got back " << refFunc2(X)
     << " from call to refFunc2(X)\n";
```

prints 6 using reference returned by refFunc2

5 Looks like passing the value of X, but refFunc2 creates a reference to X because of its argument type.

```
int& refFunc2(int& x)
{
    // Increment the variable x is referencing
    x++;
    return x;  // Return the reference
}
```

6 Return the reference. It is OK to do this because the variable which x references is outside the scope of the function.

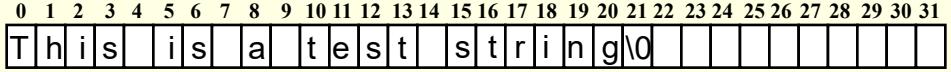
Bits and Pieces

Strings in ANSI Standard C

As implemented by St. Kerneghan and St. Ritchie

Strings shall be NULL terminated character arrays.

char str[32];



Bits and Pieces

Strings in ANSI Standard C

Header files:

#include <string.h> or **#include <cstring>**
using namespace std;

Most used functions

char *strcpy(s,ct)
char *strcat(s,ct)
int strcmp(cs,ct)

char *strncpy(s,ct,n)
char *strncat(s,ct,n)
int strncmp(s,ct,n)

char *strchr(cs,c)
char *strrchr(cs,c)
char *strstr(cs,ct)

size_t strlen(cs)

char *strtok(s,ct)

Bits and Pieces

Strings in ANSI Standard C

Header files:

#include <string.h> or **#include <cstring>**
using namespace std;

Most used functions

char *strcpy(s,ct)	char *strncpy(s,ct,n)
char *strcat(s,ct)	char *strncat(s,ct,n)
int strcmp(cs,ct)	int strncmp(s,ct,n)
char *strchr(cs,c)	size_t strlen(cs)
char *strrchr(cs,c)	
char *strstr(cs,ct)	char *strtok(s,ct)

Microsoft doesn't want you to use these.

So, tell Microsoft to...

#pragma warning(disable : 4996)

Put this at the top of each source file where you use K&R strings.

This may not work so instead do this...

1. Right click the project in Visual Studio's solution explorer pane.
2. Select Properties from the pop-up menu.
3. In the dialog box select "C/C++" then Advanced under that.
4. Click to the right of "Disable Specific Warnings".
5. Enter "4996".
6. Click the OK button.

Bits and Pieces

Strings in ANSI Standard C++

Then along came a C++ string
And sat down beside her...

Header file

```
#include <string>
using namespace std;
```

Bits and Pieces

Strings in ANSI Standard C++

Constructors:

Default constructor

```
string s1;
s1 = “Test string 1";
```

Initializer constructor

```
string s2(”Test string 2");
```

Dynamic creation of a string using default constructor

```
string *sptr1 = new string();
*sptr1 = “Test string 3";
```

Dynamic creation of a string using initializer constructor

```
string *sptr2 = new string(”testing2");
```

Bits and Pieces

Strings in ANSI Standard C++

Overloaded Operators:

Equivalence operator

=

Input/Output operators

<< >> getline(cin,str)

Logical operators

== != < > <= >=

Array operator

[ ]

Bits and Pieces

Strings in ANSI Standard C++

Changing the contents of a string

```
string& append(const string& str);
string& append(const string& str, long pos, long n);

string& assign(const string& str);

string& erase(long p0 = 0, long n = npos);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);

string& insert(long p0, const string& str);
string& insert(long p0, const string& str, long pos, long n);
void insert(iterator it, const_iterator first, const_iterator last);

string& replace(long p0, long n0, const basic_string& str);
string& replace(long p0, long n0, const string& str, long pos, long n);
```

Bits and Pieces

Strings in ANSI Standard C++

Getting size information on a string

```
long length() const;

long size() const;

long max_size() const;

long capacity() const;

bool empty() const;
```

Bits and Pieces

Strings in ANSI Standard C++

Comparing two strings

```
int compare(const string& str) const;
int compare(long p0, long n0, const string& str);
int compare(long p0, long n0, const string& str, long pos, long n);
```

Don't forget, all of these have been overloaded for string comparisons.

== != < > <= >=

Bits and Pieces

Strings in ANSI Standard C++

Find something in the string

```
long find(const string&str, long pos = 0) const;

long rfind(const string& str, long pos = npos) const;

long find_first_of(const string& str, long pos = 0) const;

long find_last_of(const string& str, long pos = npos) const;


long find_first_not_of(const string& str, long pos = 0) const;

long find_last_not_of(const string& str, long pos = npos) const;
```

Bits and Pieces

Strings in ANSI Standard C++

Getting a substring of the string

```
string substr(long pos = 0, long n = npos) const;
```

Getting a char array equivalent of the string

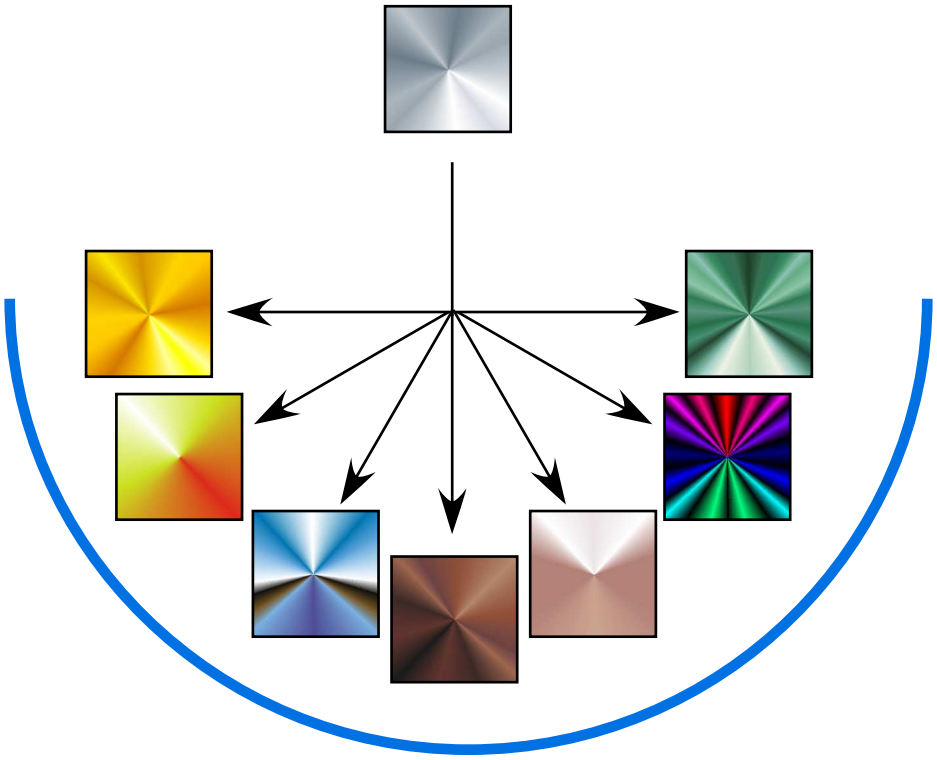
```
const char *c_str() const;
```

If you are going to use strings in your programming assignments you better remember this one. Many C/C++ and Windows OS API function calls will not accept a string object as an argument. They require a K&R char array.

Bits and Pieces

Standard Template Library

Generic Class



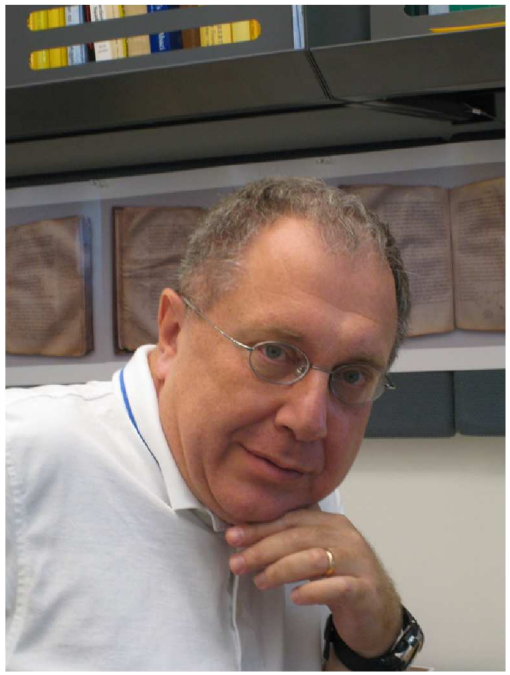
Concrete instances using different data types.

Bits and Pieces

Standard Template Library

Primarily developed by Alexander Stepanov starting in 1979

From an interview with Stepanov



Alexander Stepanov

Why did you develop the STL?

"In every programming language, there's a need for various data structures, such as vectors, lists, and associative arrays. Programmers also need fundamental algorithms -- for sorting, searching, and copying -- defined for the data structures. It has long been lamented that C++ doesn't provide a good set of standard data structures."

What has been the reaction to it?

"In the short time since its release, STL has generated many emotional -- and conflicting -- assessments. On one hand, for example, Bjarne Stroustrup **(the developer of C++)** ***of Bell Laboratories calls it a "large, systematic, clean, formally sound, comprehensible, elegant, and efficient framework."***

On the other hand, Pamela Seymour of Leiden University writes that "STL looks like the machine language macro library of an anally retentive assembly language programmer."

Bits and Pieces

Advanced C++ that is the basis for the Standard Template Library

Void pointers

```
int x = 5;
void *vptr;
vptr = reinterpret_cast<void *>(&x);
(reinterpret_cast <int *>(vptr)) = 10;
```

Function overloading

Templates produce compile time *polymorphism* while "standard" function overloading produces run-time *polymorphism.*

Function pointers

```
void myFunction(int x)
{
        cout << "x = " << x << end1;
}

void (*funcPtr)(int);
funcPtr = myFunction;
funcPtr(255);
```

```
myClass array[50];
void (*sort)(myClass[], int);
sort = BubbleSort;
sortMyClass(array, 50, sort);  // pass the sort function

template <Class T>
void sortMyClass (T *arr, int count, (void (*fp)(T [], int)))
{
   fp(arr, count);
}
```

Bits and Pieces

What is in the Standard Template Library?

- **Containers**

  Classes that hold things.

- **Algorithms**

  Generic functions that can be applied to containers to process their contents in various ways. Examples: Sort, Copy, Search, Merge, etc.

- **Iterators**

  Generalized pointer that can point to items in a container. Allows sequential access to all items in a container.

Bits and Pieces

What is in the STL? A closer look.

Containers - Classes that hold things.

- **Ordered Collections**

  vector, list, deque

- **Unordered Collections**

  set, multiset, map, multimap
  (hash_set, hash_multiset, hash_map, hash_multimap)

- **Other Containers**

  bitset, valarray

Container Adapters - Classes that modify the functionality of containers.

stack, queue, priority_queue

Bits and Pieces

Ordered Containers

<vector>

- Random access
- Dynamic array of objects
- Insert/remove at end in O(1)
- Insert/remove at beginning or middle in O(n)
- Special implementation for type bool

Defined in header:*

```
#include <vector>
```

Sample Constructors:

```
vector<double>dVec;
vector<int>iVec(5,0);
vector<float>fVec(20);
vector<string>sVec(10, "abc");
```

* using namespace std; is required for all template classes.

Bits and Pieces

Ordered Containers: *vector*

Most Used Functions

Changing the contents

- push_back(newValue)
- insert(itr, newValue)
- pop_back()
- clear()
- erase(itr1, itr2)
- resize(n)

Getting size information

- size()
- capacity()
- max_size()
- empty()

Locating elements

- *v*[*n*]
- front()
- back()

Iterators

- begin()
- end()
- rbegin()
- rend()

Related Template Functions

- swap(v[x], v[y])

Bits and Pieces

Ordered Containers

<list>

- Created as a double linked list
- Elements not stored contiguously in memory
- Slow lookup and access (linear time)
- Quick insertion and deletion (constant time)

Defined in header:*

```
#include <list>
```

Sample Constructors:

```
list<double>dList;
list <char>cList ();
list <simple>simList ();
```

* using namespace std; is required for all template classes.

Bits and Pieces

Ordered Containers: *list*

Most Used Functions

Changing the contents

- push_back(newValue)
- push_front(newValue)
- insert(itr, newValue)
- remove(T)
- assign(n, T)
- pop_back()
- pop_front()
- clear()
- erase(itr1, itr2)
- sort()

Getting size information

- size()
- capacity()
- max_size()
- empty()

Locating elements

- front()
- back()

Iterators

- begin()
- end()
- rbegin()
- rend()

Bits and Pieces

Ordered Containers

<deque>

- Created as a double linked list
- Similar to vector
- Iterators may become invalid
- Quick insertion and deletion (constant time) at beginning or end.
- Random inserts allowed, but inefficient.

Defined in header:*

```
#include <deque>
```

Sample Constructors:

```
deque<double>Deque;
deque <char>Deque ();
deque <simple>simDeque ();
```

* using namespace std; is required for all template classes.

Bits and Pieces

Ordered Containers: *deque*

Most Used Functions

Changing the contents

- push_back(newValue)
- push_front(newValue)
- insert(itr, newValue)
- remove(T)
- assign(n, T)
- pop_back()
- pop_front()
- clear()
- erase(itr1, itr2)
- sort()

Getting size information

- size()
- capacity()
- max_size()
- empty()

Locating elements

- d[n]
- front()
- back()

Iterators

- begin()
- end()
- rbegin()
- rend()

Bits and Pieces

Containers Adapters

<stack>

- LIFO structure
- Wraps a <deque> template
- Provides push and pop functionality
- No other access is allowed

Defined in header:*

```
#include <stack>
```

Sample Constructors:

```
stack<dataType>  aStack;
stack<int> iStack;
stack<char> cStack;
```

* using namespace std; is required for all template classes.

Bits and Pieces

Containers Adapters: *stack*

All Available Functions

Changing the contents	Locating elements	Getting size information
push(val) pop()	top()	size() empty()

Bits and Pieces

Containers Adapters

<queue>

- FIFO structure
- Wraps a <deque> template
- Provides push and pop functionality
- No other access is allowed

Defined in header:*

```
#include <queue>
```

Sample Constructors:

```
queue<dataType>  aQueue;
queue<int> iQueue;
queue<char> cQueue;
```

* using namespace std; is required for all template classes.

Bits and Pieces

Containers Adapters: *queue*

All Available Functions

Changing the contents	Locating elements	Getting size information
push(val) pop()	front() back()	size() empty()

Bits and Pieces

Containers Adapters

priority_queue

- Works like a queue except that pop() removes the item with the highest priority (value).
- Wraps a <list> template
- Provides push and pop functionality
- No other access is allowed

Defined in header:*

```
#include <queue>
```

Sample Constructors:

```
priority_queue<dataType>  paQueue;
priority_queue<int> piQueue;
priority_queue<char> pcQueue;
```

* using namespace std; is required for all template classes.

Bits and Pieces

Containers Adapters: *priority_queue*

All Available Functions

Changing the contents	Locating elements	Getting size information
push(val) pop()	top()	size() empty()

Bits and Pieces

Iterators

```
vector<MyClass> m_vMyStuff;
// Fill vector with instances of MyClass

for(vector<MyClass>::iterator itr = m_vMyStuff.begin();  // Loop counter
            itr != m_vMyStuff.end();                     // Loop test
            itr++)                                       // Loop increment
{
      cout << itr->m_iItemID << endl;       // Use itr as pointer to instance
      cout << itr->m_sDataStr << endl;      //      of MyClass

}
```

```
struct simple
{
    int x;
    char ch;
};
```

Bits and Pieces

Collections of pointers to dynamically created objects

```
vector<simple *> SS;
simple *temp;

temp = new simple();
temp->x = 1;
temp->ch = 'A';
SS.push_back(temp);
// Do the same for several other instances of simple
for(i=0; i < SS.size(); i++)
{
  cout << "x = " << SS[i]->x << "  ch = " << SS[i]->ch << endl;
}
for(vector<simple *>::iterator cii = SS.begin(); cii != SS.end(); cii++)
{
  cout << "x = " << (*cii)->x << "  ch = " << (*cii)->ch << endl;
}
```

*Remember the iterator is a pointer to an object in the vector. In this case that object is a pointer. Thus, you must dereference the iterator (*cii) to get the pointer then dereference that to access the instance of struct simple.*

```
struct simple
{
    int x;
    char ch;
};
```

Bits and Pieces

Collections of dynamically created objects

```
vector<simple> SS;
simple *temp;

temp = new simple();
temp->x = 1;
temp->ch = 'A';
SS.push_back(*temp);
// Do the same for several other instances of simple
for(i=0; i < SS.size(); i++)
{
  cout << "x = " << SS[i].x << "  ch = " << SS[i].ch << endl;
}

for(vector<simple>::iterator cii = SS.begin(); cii != SS.end(); cii++)
{
  cout << "x = " << (cii)->x << "  ch = " << (cii)->ch << endl;
}
```

You must dereference the pointer to push the object into the vector.

You have the object here so you must use dot notation to access the fields.

Here, cii is a pointer to an instance of simple so use pointer notation to access the fields.

Bits and Pieces

The use of *goto*

Just don't do it!!!

```
for(int i=0; i<9999999; i++)
{
        cout << "i is now " << i;
        if(i > 10)
            goto bail;
}

bail:       // a label
cout << "Exiting the dumb loop.";
```