Pointers

What are pointers?

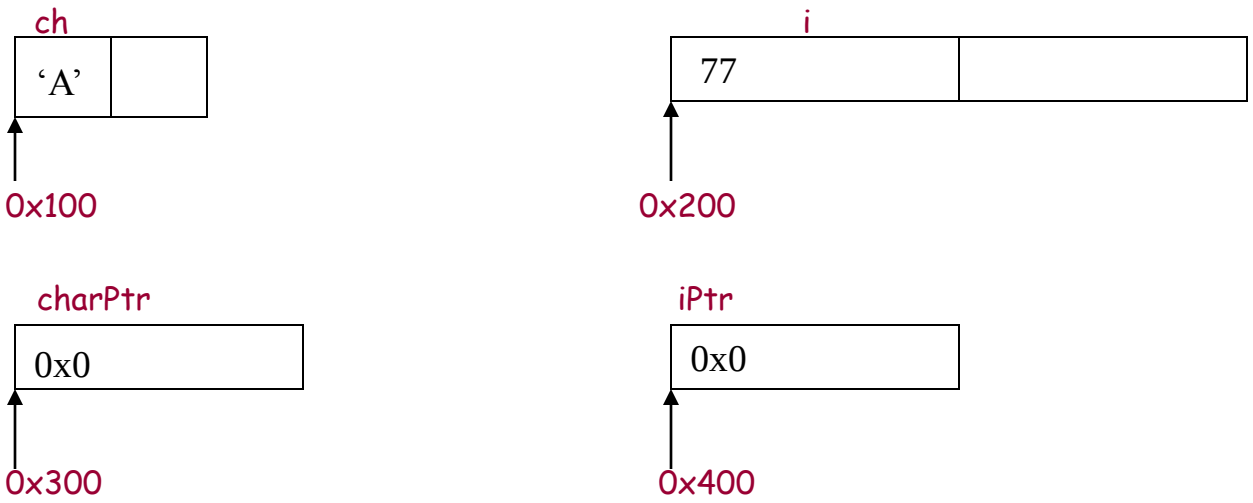
- Pointers are variables that hold the address of another variable (or object).
- A pointer variable is said to be "pointing" to this object/variable.

Example

- Lets say we have the following definitions:

```
char ch = 'A';
int  i = 77;
char * chPtr = NULL;
int * iPtr = NULL;
```

This will look something like the following:

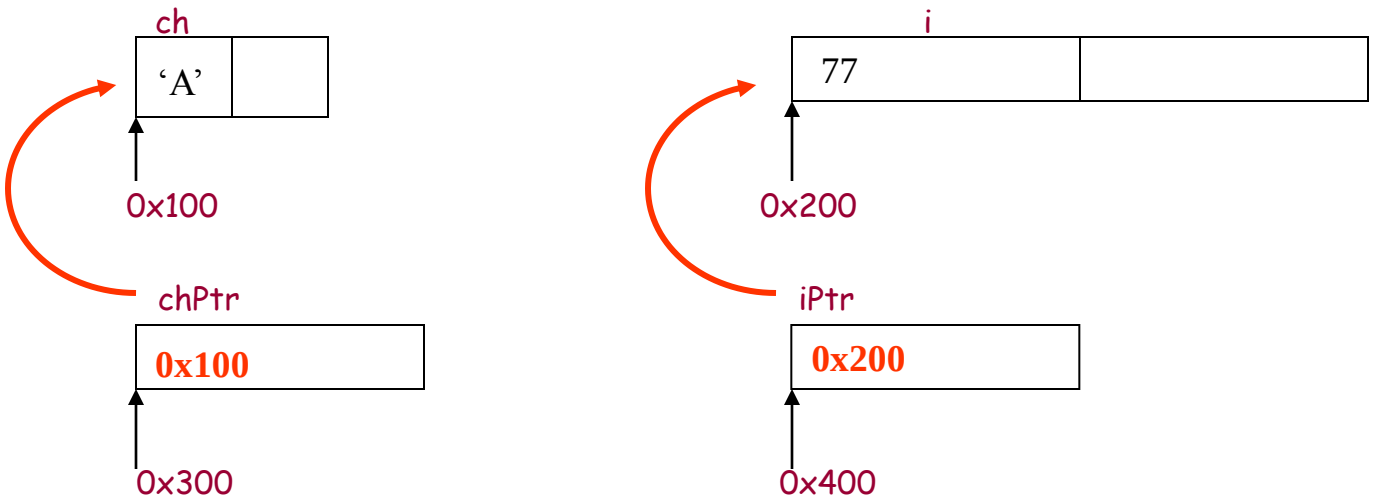


Example (contd.)

- Lets say we do the following assignments:

```
chPtr = &ch;
iPtr = &i;
```

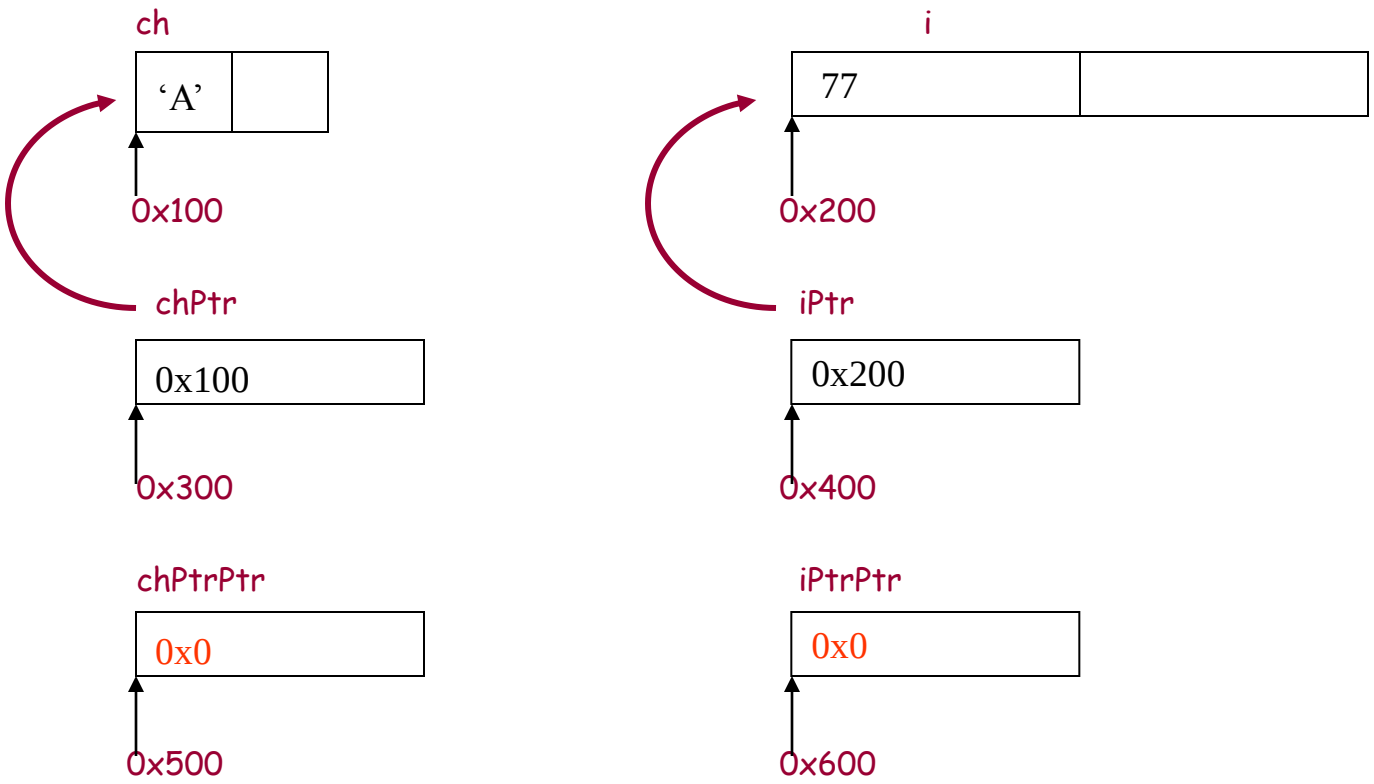
This will look something like the following:



Pointer to pointer

- Now, lets say we have the following two definitions also:

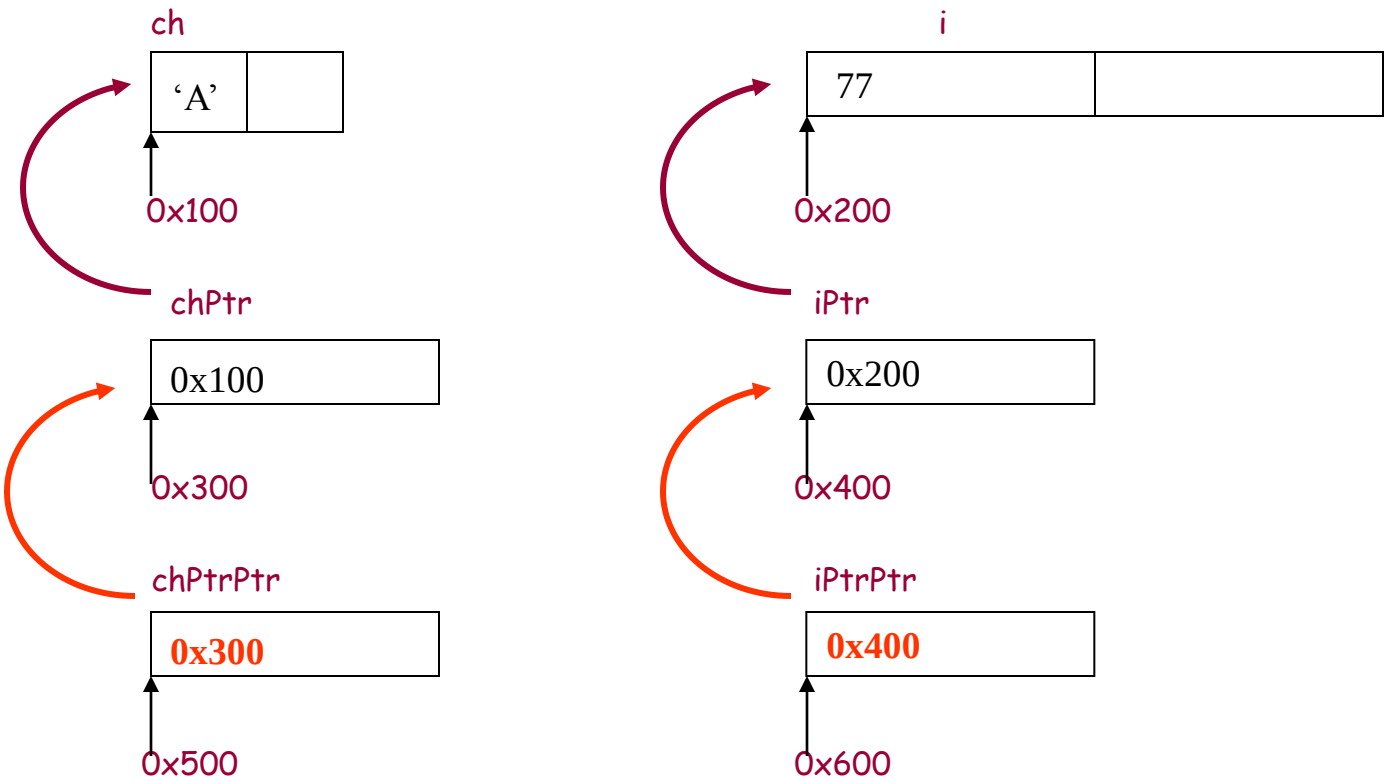
```
char * *    chPtrPtr = NULL;
int * *     iPtrPtr = NULL;
```



Example (contd.)

- Now, lets say we do the following assignments :

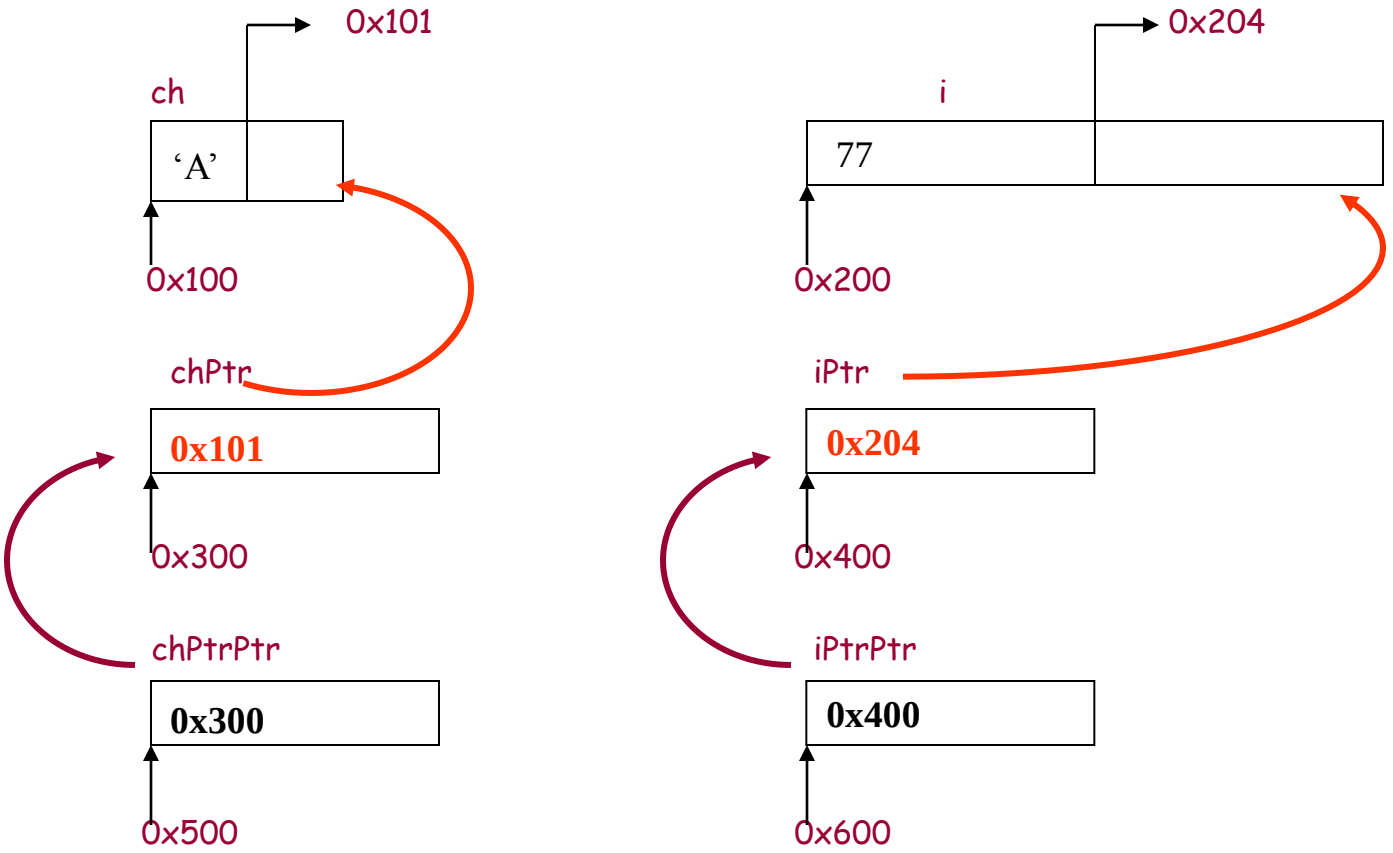
```
chPtrPtr = &chPtr;
iPtrPtr = &iPtr;
```



Example (contd.)

- Now, lets say we do the following changes :

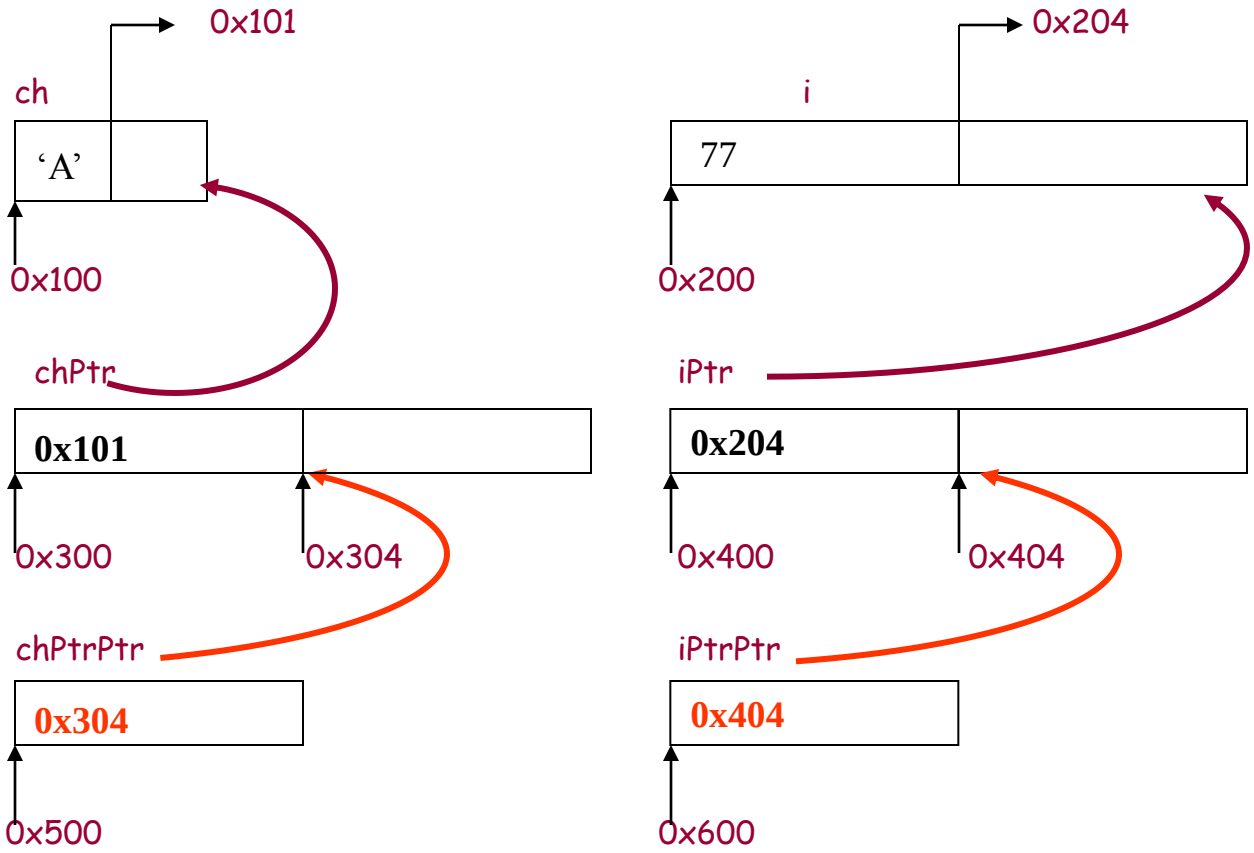
```
chPtr ++;
iPtr ++;
```



Example (contd.)

- And, if we do the following changes :

```
iPtrPtr ++;
chPtrPtr ++;
```

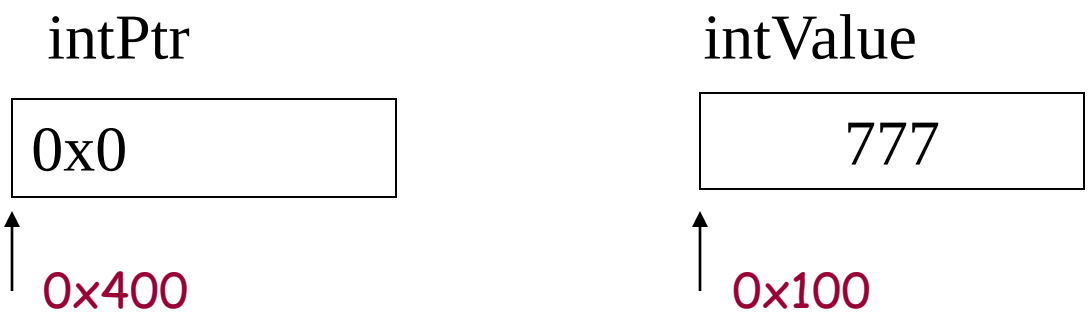


Another example

Consider the following two statements :

1) int intValue = 777;
2) int * intPtr = NULL; // intPtr points to nothing

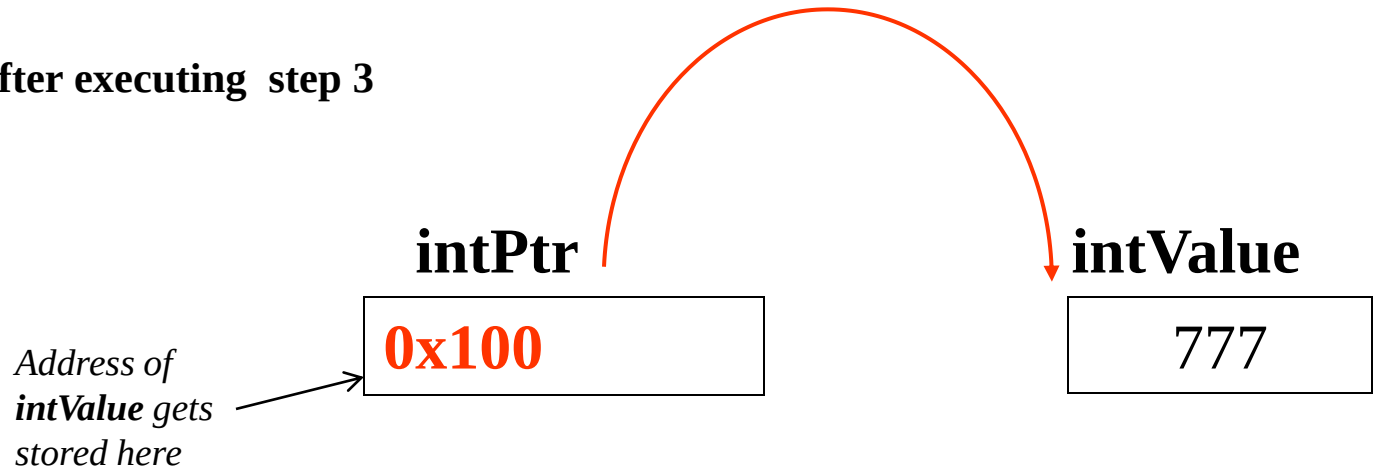
After executing steps 1 and 2



contd.

```
3)   intPtr = & intValue;        //  intPtr points to intValue
4)   *intPtr = 999;              //  intValue now contains 999
```

After executing step 3



After executing step 4

So, what are pointers again?

- Pointer is a means of having an alias to an object.
  - Example:
    - ```
      int   ii = 2;
      int   * ptr = & ii;          // *ptr is 2.   ptr is an alias to ii
      ii = 4;                      // *ptr is 4
      *ptr = 5;                    // value of ii  is 5
      ```

- Pointers are used when passing an argument by address.
  - Example:
    - `void  foo  ( MyObject * objPtr )` // foo expects a pointer
    - Calling this function : `foo ( & myObj )` // pass address of myObj
      ```

String literals

- String literals can be assigned to a pointer to a *char*.

- A string literal is a character sequence in double quotes:
  - "Here is a string literal"
  - "OGI 505" // another string literal

  - char * ptr = "OGI 505";
  - ptr[1] = 'C'; // ERROR. Cannot change string literals

  - really should say const char * ptr = "OGI 505";

  - That way ptr [1] = 'C'; is a compile error

String literals

- It is ok to return them from a function since they are statically allocated.

- ```
  const char * GetHelpString ( int helpStringNum )
  {
        …
        return  "Help String A";
  }
  ```

- Initializing long strings: You can break up the initialization to make it readable.

  - ```
    char  longString1  [ ] = "This is an attempt"
                             "at a long string ";
    ```

  - ```
    char   longString2 [ ] = "Putting a new line
                             is not allowed";                    // ERROR
    ```
    ```

Array name

- An array name is a pointer to its first element, i.e., the address of its first element.
  - Example:
    - int integerArray [20];
    - int * intPtr;
    - intPtr = integerArray; // intPtr points to first element
      - or
    - intPtr = & integerArray [0]; // intPtr points to first element
  - Above two assignments to intPtr are equivalent.
  - If we say
    - *intPtr = 100;
  - We are essentially saying
    - integerArray [0] = 100

Array indexing

- Since the array name is a pointer to the first element, we can access other elements by incrementing (or decrementing a pointer).

  - Example:
    - int intArray [20];
    - int *intPtr = intArray;

    - intPtr + 1 is address of 2nd element (i.e., & intArray[1])
    - intPtr + 19 is address of 20th element (i.e. at index 19)
    - intPtr + 20 is one past the last element in the array and accessing this or assigning anything to this address is a runtime **error**.

    - *(intPtr + 1) = 200; // equivalent to intArray[1] = 200
    - *(intPtr + 19) = 400; // equivalent to intArray[19] = 400
    - *(intPtr + 20) = 500; // equivalent to intArray[20] = 500
      // This last stmt is an **ERROR**

Pointer arithmetic

- When you increment or decrement pointers, they increment or decrement by the size of what they are pointing to.

- Example:

  - char * charPtr = new char [10];
  - Lets say charPtr contains address 0x100
  - (charPtr + 1) is 0x101, i.e., it increments by 1 byte, i.e., sizeof (char)

  - int * intPtr = new int [10];
  - Lets say intPtr contains address 0x200
  - (intPtr + 1) is 0x204, i.e. it increments by 4 bytes, i.e. sizeof (int), assuming of course that sizeof (int) is 4.

  - Because of this, we can access elements of an array by incrementing the pointer to its first element.

Pointers to various types

- Pointers to built in types:
  - int * intPtr;
  - char * charPtr;
  - double * doublePtr;
- Pointers to pointers and pointers to arrays:
  - char * * ptr; // ptr is pointer to pointer to a char
  - int * ptr2 [10]; // ptr2 is an array of integer pointers
  - int (* arr) [ ] // arr is a pointer to an array of integers

Pointers to various types

- Pointers to functions:

- `int ( * func1 ) ( );` // func1 is pointer to a function returning an int

- `int ( * func2 ) (const char *);` // func2 is pointer to a function
  // returning an int, and taking a
  // const char * as an argument

- `int * ( * func3 ) (const char *);` // same as above, except returns
  // a pointer to an int

- Following are simple function declarations:

- `int func4();` // func4 is a function returning an int

- `int * func5();` // func5 is a function returning a pointer
  // to an int

The void pointer

- While we cannot have an object (or variable) of type void, we can have a pointer to void. So,
  - Illegal: void v;
  - Legal: void * ptr;

- A void * is ok to have since it is a pointer, and all pointers have the same size, i.e., whether we have a pointer to an int, or a char, or a double, or a void, the pointer itself will be the same size (4 bytes on a 32 bit system).

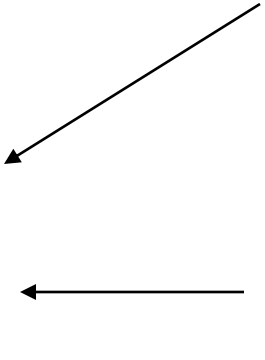
- A void * is essentially a data storage thing, i.e., we can only store into a void pointer, not dereference it, since it does not have any type information.
- If we have to dereference, we need to cast it to the proper type.

- So, *ptr would be an error, but, say we stored the address of an int in a void pointer, we could say * (int *)ptr, i.e., cast it to an int *, then dereference.

Stack variables

- Memory allocation for stack variables is handled by the compiler.
- This memory is statically allocated by the compiler since the size is known at compile time.
- As a result, this memory is released by the compiler when the function exits.
- Example: All of the following local variables are stack variables.

```
void      foo ( char * arg1 )
{
   int              ii;
   char             ch;
   MyClass          myObj;
   int  *           intPtr;
   MyClass  *       myObjPtr;
   ...
}
```



Stack objects, memory for these is on the stack.

The pointers don't point to anything valid yet, can be made to point to memory allocated on the heap.

NOTE: Pointers themselves are on stack.

Stack variables

- Stack variables get deleted even if you have multiple returns from the function, or if the function throws exceptions.

- Example:
  -
```
int   foo ( char * arg1 )
{
   int  ii;
   MyClass  mcObj;
   char   chArray[ 100 ];
   …
   if ( someCondition )
          return  5;                    <<-- Exit point
   else if ( someOtherCondition )
          return 6;                     <<-- Exit point
   else
          throw  someException;         <<-- Exit point
}
```

Heap

- Heap memory is allocated at runtime by the programmer, and should be deleted when it is no longer needed.
- If this memory is not explicitly deleted by the programmer, there is a memory leak.
- Have enough of these memory leaks and your program can become really slow.
- There are tools available to check for memory leaks (Purify from Rational Software, BoundsChecker).

Example

```
- void        foo ( char * arg1 )
  {
     int              ii;
     int  *           intPtr;
     MyClass  *       myObjPtr;

     intPtr = new  int ( 10 );              // allocate an int, assign initial value 10
     myObjPtr  =  new  MyClass ( );         // allocate a MyClass object and use
                                            // default constructor (if available)
                                            // to initialize the object.

     // when done, make sure it is deleted, unless you want to return this memory
     // from the function

     delete  intPtr;                        // releases memory for the int
     delete  myObjPtr;                      // releases memory for MyClass object
                                            // after calling MyClass destructor
  }
```

Another example

```
int   foo ( char * arg1 )
{
    int  ii;
    MyClass  * mcPtr  =  new  MyClass();                  // create on heap
    char   * chArray = new  char [ 100 ];                 // create on heap
    …
    if ( someCondition )
    {
      delete mcPtr ;   delete  [ ] chArray;               // Delete heap memory
      return  5;                        <<--  Exit point
    }
    else if ( someOtherCondition )
    {
      delete mcPtr ;   delete  [ ] chArray;               // Delete heap memory
      return 6;                         <<--  Exit point
    }
    else
      throw  someException;         <<--  Exit point   // Oops, forgot to delete.
}
```

Some points

- It is ok to call the *delete* operator on a null pointer:
  - This means you don't have to check:
    - if (intPtr != NULL)
      delete intPtr;
      OR
    - if (myObjPtr != NULL)
      delete myObjPtr;
  - Instead, can say:
    - delete intPtr;
    - OR
    - delete myObjPtr;
  - If the pointer is NULL, calling *delete* on it is a no-op.

Some more points...

- Can point to stack variables as long as the pointed to value is not accessed *after* the stack variable has gone out of scope.
- As a result, a function should not return pointers to stack variables.

- Example:

```
char *  MyFunction ( char *  ptr )
{
    int * intPtr = NULL;

    if (someCondition)
    {
        int  ii = 100;
        intPtr = & ii;
        ...
    }

    // Illegal to dereference  intPtr here since stack variable ii has gone out of
    //scope.

}
```

Another example

- Can you spot the bugs in the code below:

```
char *  MyFunction ( char *  ptr )
{
    char  array[ 10 ] = "Testing";
    char  * returnValue;

    sprintf( ptr, "MyFunction returns :  %s %d\n", array, 200 );

    returnValue =  array;      // point to  beginning of array which contains "Testing"
    …

    return  returnValue;
}
```

Garbage collection

- There is no garbage collection in C++.
- This means you are responsible for managing the allocation and deallocation of heap memory.
- A program should not leak memory :-).
- Java has garbage collection. As a result, you don't have to worry about deleting anything. Just use it and forget it.
- When some memory is no longer pointed to, it will get marked for deletion. Note that this does not mean it will get deleted immediately, but next time when garbage collection kicks in.

Reference

- A reference is an alias (another name) for an object.

- It can be used for passing arguments to a function, or returning values from a function or overloaded operators.

- When you declare a reference, you have to initialize it also.
- This also ensures that a reference is bound to something.
- In C++, a reference cannot be bound to nothing.

- Example:
  - `int &  intRef1;  <-- Error, needs to be initialized also`
  - `int  int2;`
  - `int &  intRef2 = int2;  <-- this is ok, bound to int2`

  - `extern int &  extInt;  <-- this is ok too.`

Reference

- Once a reference is initialized, you cannot change it,

- Example:
  - int int1 = 100;
  - int int2 = 200;
  - int & intRef2 = int2; <<-- intRef2 bound to int2

  - intRef2 = int1; <<-- intRef2 still bound to int2

  // Above statement assigns the value of int1 to the object that intRef2 is bound to.

  // So, int2 is now 100.

  - intRef2 ++; <<-- intRef2 is 101, so is int2.
    <<-- int1 is still 100

Reference

- Since a reference is like a pointer, it needs to be bound to an lvalue.
- Following are errors:
  - int & intRef = 22;
  - char & charRef = 65;
- However, the following are ok:
  - const int & intRef = 22;
  - const char & charRef = 65;

Usage

- References can be used to pass arguments to functions where the function intends to change the value of the argument.

- Example:
  - ```
    void  GetValues ( int &  count,  double &  average )
    {
      count = 100;
      average = 151.2;
    }
    ```

  In C, you would have passed these arguments by address.
  - `void GetValues ( int * count, double * average )`

  - And then would have to first check if they are not NULL.
    ```

Potential bugs with pointer usage

- Dangling pointer:
  - A dangling pointer is one that is pointing to memory that has already been deleted. This *may* cause a run time error (access violation) if you are lucky.

  - Example:

```
char * ptr1 = new char [ 10 ];
char * ptr2 = ptr1;
delete  [ ] ptr1;
*( ptr2 + 1 ) = 'A';          <<-- ERROR !  Using dangling pointer
```

- Another example of dangling pointer is on next slide where the copy constructor does a shallow copy, and destructor releases the memory.

Example:

```
class classA  {
    public:
                classA ( const char * str );          // conversion
                classA ( const classA &  other );     // copy
                ~classA ();                           // destructor
                ...
    private:
                char * m_Str;
}

classA::classA ( const char * str )
: m_Str (NULL)
{
   if ( str)  {
      m_Str = new char [ strlen (str) + 1 ];          // allocate memory
      strcpy ( m_Str,  str );    }
}

classA::classA ( const classA & other )
{     m_Str  =  other.m_Str;  }                       // shallow copy

classA::~classA()
{     delete [] m_Str;     }                    // release the memory allocated
```

Example contd.

- void foo1 (const classA & objRef)
  { … }
- void foo2 (classA obj)
  { … }

- main()
  {
  classA clA ("Testing");

  foo1 (clA); // clA passed by reference. No problem here
  foo2 (clA); // clA passed by value, temporary created and
  // passed to foo2.

  // At this point, clA.m_Str is a dangling pointer.
  }

Moral of the story : Copy constructor should not do a shallow copy if the pointer is being deleted in the destructor.

Double deletion

- Double deletion:
  - When you delete the same memory more than once. This can cause a run time error.
  - Extending the previous example:
  -
```
main()
{
   classA  clA1 ( "Testing" );
   classA  clA2 ( claA1 );
   …
}          <<-- At this point clA1 is destructed, deleting clA1.m_Str.
                And then clA2 is destructed, deleting clA2.m_Str which
                is pointing to deleted memory, thereby doing a double
                deletion.
```

Passing pointers by reference

- If you want to change the value of the pointer inside a function, you need to pass the pointer itself by reference (or by address)
- Example:

```
void  FreeMem ( int * ptr )          // ptr is passed by value
{
   delete  ptr;
   ptr = NULL;                       // for safety.   ptr is now NULL
}

int main()
{
   int  * intPtr = new int [ 10 ];   // Lets say intPtr = 0x100
   // use intPtr

   FreeMem ( intPtr );

   // intPtr is  NOT  NULL  at this point.  It is still  0x100, although the
   // memory has been deleted, so we cannot access it.
}
```

Example continued

Fix is to declare FreeMem as:

```
void   FreeMem ( int * &  ptr )
{
    delete  ptr;
    ptr = NULL;                    // for safety.   ptr is now NULL
}
```

OR the C- style:

```
void   FreeMem ( int **  ptr )
{
    delete *ptr;
    *ptr = NULL;
}
```

in this case the call looks like FreeMem (&intPtr);

Smart Pointers

- Smart pointers know when and how to delete what they are pointing to.
- They can do other things in addition to just deletion (depends on what is implemented).
- For deletion, smart pointers depend on the fact that they themselves will get deleted on a function exit (since we create them on stack).
- When a smart pointer created on the stack is deleted, its destructor is called, and inside the destructor, it will delete what it is pointing to.

Example using smart pointers

```
int   foo ( char * arg1 )
 {
    int  ii;
    auto_ptr < MyClass > mcPtr (  new  MyClass() );          // create on heap, using smart ptr
    auto_ptr < char > chArray ( new  char [ 100 ] );         // create on heap, using smart ptr
    …
    if ( someCondition )
    {
      // delete mcPtr ;   delete  chArray;                   // No need to call delete
      return  5;                          <<--  Exit point
    }
    else if ( someOtherCondition )
    {
      // delete mcPtr ;   delete  chArray;                    // No need to call delete
      return 6;                           <<--  Exit point
    }
    else
      throw  someException;               <<--  Exit point   // No need to call delete
 }
```