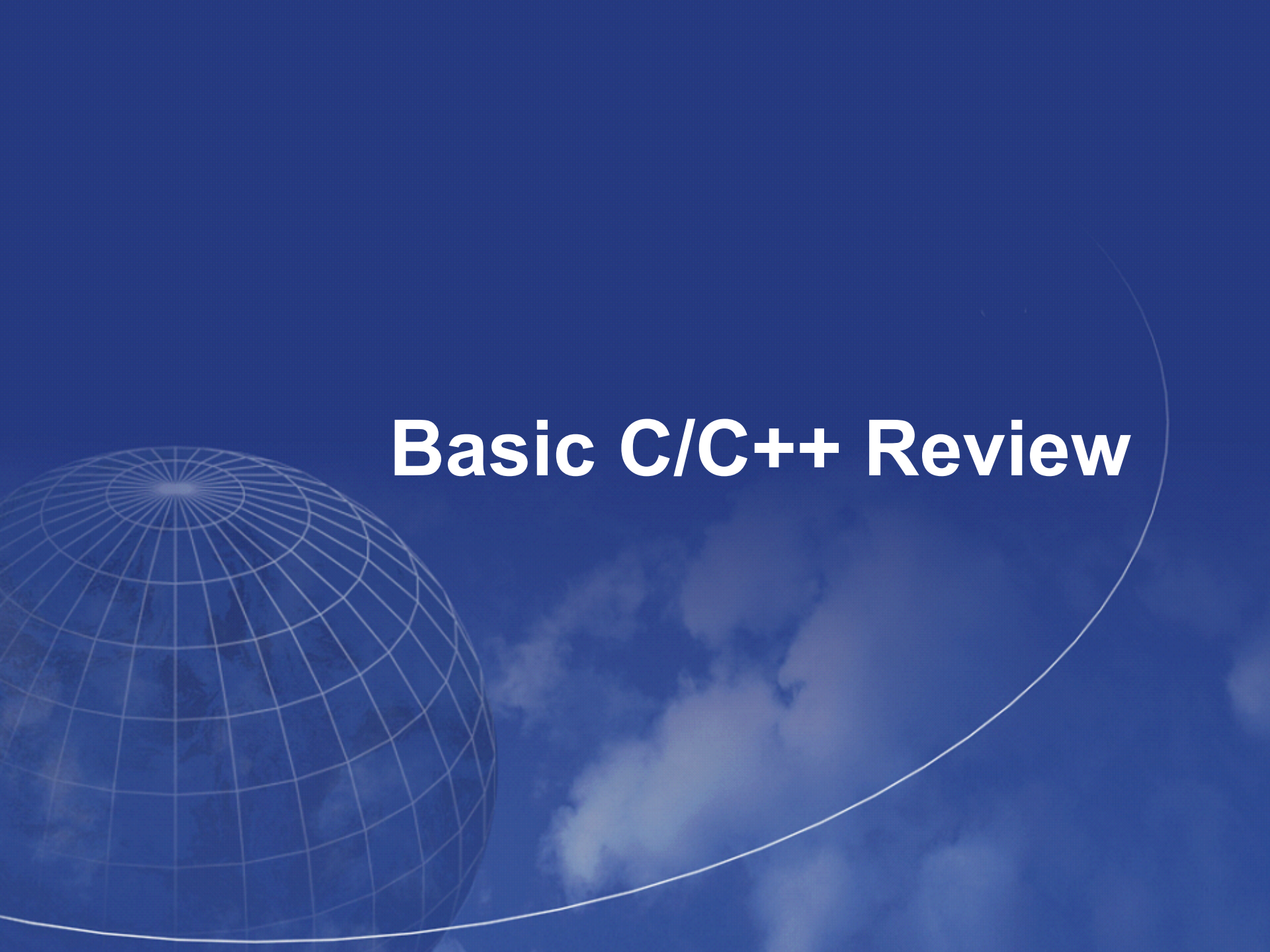

Basic C/C++ Review

C/C++ Language Review

- Enumerated types
- Bitfields
- Bitwise Operators
- Typedefs
- Typecasting
- Pointers and Addresses
- Structures and Unions
- Type qualifiers
- Arrays
- Stacks & Heaps

Enumerated Types

- An enumeration is a set of named integer constants.
- A variable of and enumeration type stores one of the values of the defined enumeration set.
- Variables of `enum` type can be used in indexing expressions and as operands of all arithmetic and relational operators.
- Enumerations provide an alternative to the `#define` preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

*More about this later

Enumeration Examples

```
enum BOOLEAN {false, true};
enum DAY {
   saturday,  sunday = 0,
   monday,    tuesday,
   wednesday,      thursday,
   friday
} weekday;
```

- The value 0 is associated with `saturday` by default. `Sunday` is explicitly set to 0. The remaining identifiers are given the values 1 to 5 by default.
- To set the variable `today` to a value from the set `DAY`.

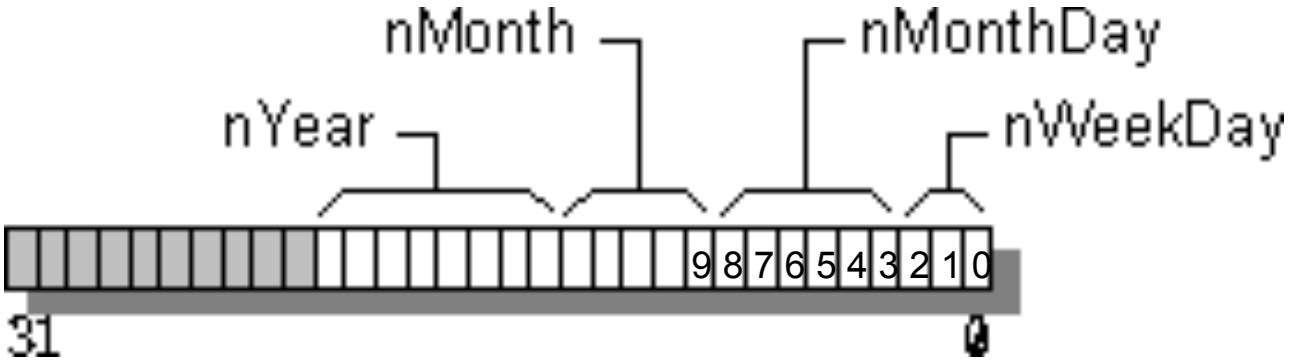
```
enum DAY today = wednesday;
```

Bitfields

- These are structures having members that occupy less storage than an integral type.
- Bitfield members are defined by a *type-specifier* that must be either `unsigned int`, or `int`.
- The width of each field in bits must be a non-negative integer value

```
struct Date {
   unsigned nWeekDay   : 3;    // - 0..7
   unsigned nMonthDay  : 6;    //- 0..63
   unsigned nMonth     : 5;    //- 0..31
   unsigned nYear      : 8;    //- 0..255
} dbits;
```

Bitfields



- Memory layout of bitfield
- We can access each field as :

```
dbits.nYear = 156;
dbits.nMonth = 5;
```

- Bit fields must be long enough to contain the bit pattern. For example, the following is not legal:

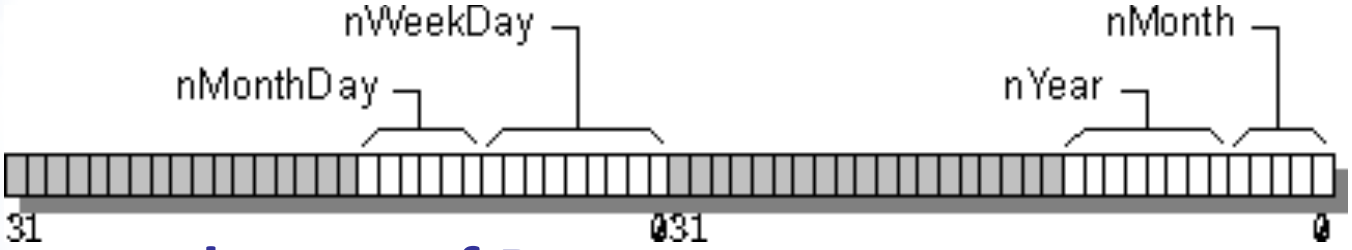
```
short a:17;          //- Illegal!
long  y:33;          //- Illegal!
```

Unnamed Bitfields

- The name (or *declarator)* of each member can be either named or anonymous. Anonymous bit fields are those with no identifier and can be used for padding.
- An unnamed bit field of width 0 forces alignment of the next bit field to the next *type* boundary, where *type* is the type of the member.

```
struct Date {
   unsigned nWeekDay   : 3; //- 0..7
   unsigned nMonthDay  : 6; //- 1..63
   unsigned            : 0; //- alignment
   unsigned nMonth     : 5; //- 0..31
   unsigned nYear      : 8; //- 0..255
};
```

Bitfields Restrictions



- Memory layout of Date structure
- Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed.
- Bit fields can only be declared within a structure.
- The address-of operator (&) cannot be applied to bit-field components.
- Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable.

Bitwise Operators

- These operate on integers as if they were a series of bits rather than whole numbers

Operation	C	C++
Complement	~	~=
AND	&	&=
Exclusive OR	^	^=
Inclusive OR	|	|=
Shift Left	<<	<<=
Shift Right	>>	>>=

A	B	~A	A & B	A ^ B	A | B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

Bitwise Operators Example

- Given A = 0000111,

  B = 00111100

A & B	00000100
A ^ B	00111011
A | B	00111111
~(A | B)	11000000
~A & B	00111000
A << 3	00111000
B >> 3	00000111

Bit Masks

- Bitwise operators are often used to create and apply ***masks***. These are constants or variables that are used to extract bits from other variables or expressions.
- EG. Extract the high order byte from a 16 bit word

```
(var & 0xFF00) >> 8;        //=0x00XX
(var >> 8) & 0x00FF;        //=0x00XX
var & 0xFF00;               //=0xXX00
```

- EG. Extract the middle 4 bits from a byte:

```
var &= 0x3C;                //- 0x3C = 00111100
```

Bit Packing

- Bitwise operators can be used to pack bits across byte boundaries.
- To pack bytes into an 32 bit int we can do:

```
long   num = 0;
char   a = 0x0A, b = 0x40;
char   c = 0xC0, d = 0xFF;
num = (num << 8)| a; //= 0x0000000A
num = (num << 8)| b; //= 0x00000A40
num = (num << 8)| c; //= 0x000A40C0
num = (num << 8)| d; //= 0x0A40C0FF
```

Ternary Conditional Operator

- A ? B : C
- This means: If A them B else C
- Examples:

```
int num = 3;
int max = num > 2 ? 3 : 4;          //- max = 3

int num = -5;
int abs = num < 0 ? -num : num;
```

Typedefs

- These are used to create new data type definitions.
- You can use `typedef` declarations to construct shorter or more meaningful names for existing types or for types that you have declared.

```
typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef unsigned long    DWORD;
typedef BYTE*            BYTEP;
typedef struct PT {int x,y;} POINT;
```

- Now we can declare the following variables:

```
POINT  coords = {1,3};
BYTE   letter = 'a';
```

Typecasting

- Used to change the data type of a variable.

```
char character = 'a';
long n = (long)character;

char   array[10] = "string";
void* p = (void*)array;
```

Pointers

- **Pointer variable:**
  - A variable that stores a memory address
- **Declaration Syntax:**

  dataType *identifier;
- **Examples**
  - `int  *pa;  int* pb;  int *  pc;`
  - `char* pc;  int *p,q; float *p, *q;`
- **We need to initialise pointers with the address of something (never leave one uninitialised):**

```
Pointer = NULL;
Pointer = &Variable;
Pointer = Array_name;
Pointer = &Array_name[0];
```

Pointers and Addressing

- The '&' operator gives the address of an object.

```
int X;
int* pX = &X;
int** ppX = &pX;
```

```
int arr[5];
int* p = arr;
int* q = &arr[3];
```

- "Dereferencing" a pointer is when we refer to what a pointer "points to" by putting a '*' in front of its name :

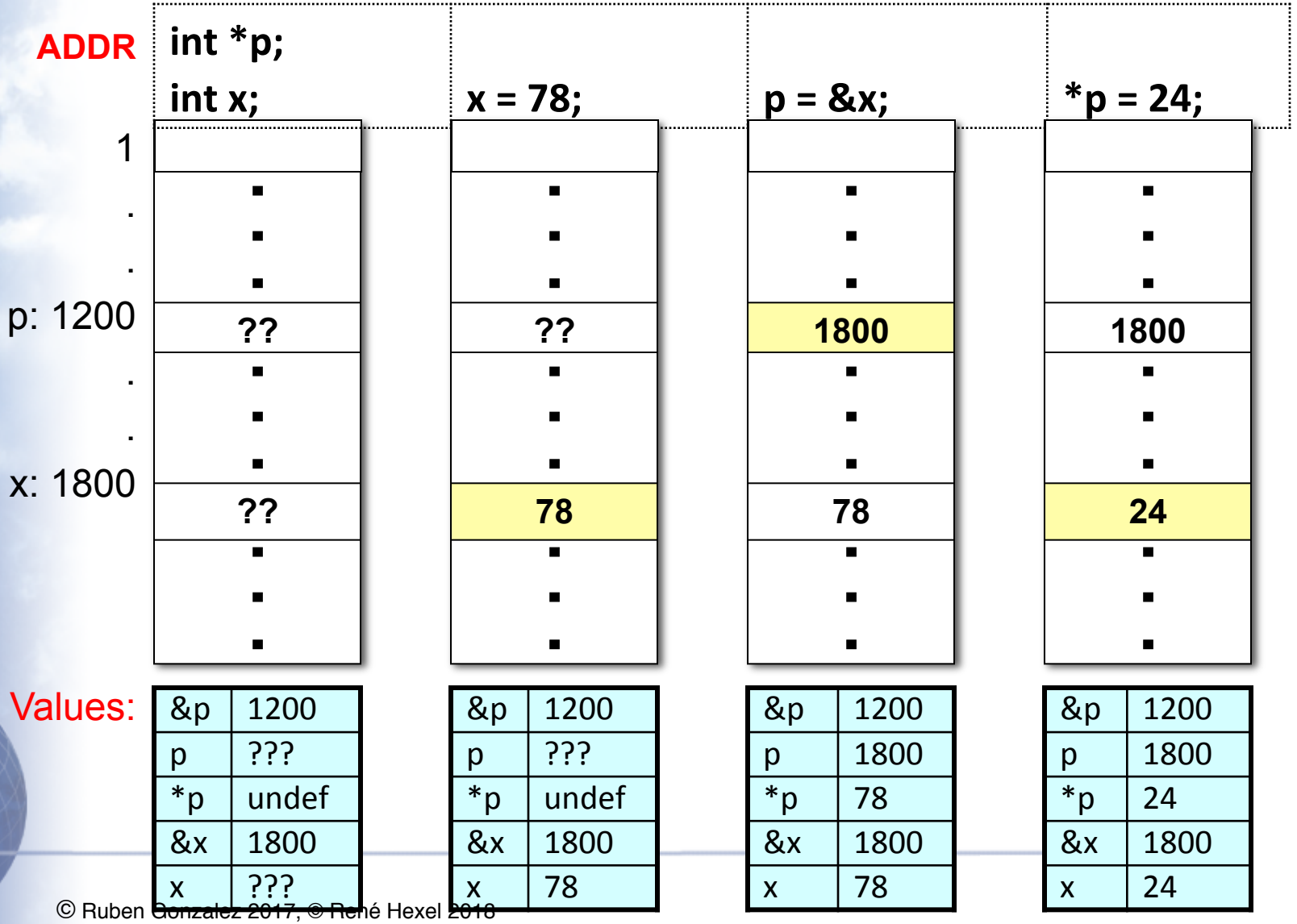
```
value = *Pointer;
```

- Summary of Pointer related operators

Address of operator: &	*Indirection operator: **
`int x;` //- variable	`int x = 25;` //- variable
`int *p;` //- pointer	`int *p = &x;` // p = addr x
`p = &x;` //- store addr of x	`*p = 40;` // x = 40

Pointers

- Consider the effect of program statements in main memory



Pointers

- Summary of pointer operations
  - &p, p, and *p all have different meanings
  - &p means the address of p
  - p means the value of p
  - *p means the value pointed to by p, ie pointed to by the value of p which is a memory location
- WARNING:
  - Make sure you don't use an uninitialised pointer!
  - Example

```
int * p;
int n = *p; //- *ptr not initialised
```

  - Intialisation

```
char *p = NULL;
char *p = 0;
```

Operations on pointer variables

```
int *p, *q;
double *d;
```

- **Assignment operations**

```
p = q;  //- makes p point to the same location as q
```

- **Relational operations**

```
p == q; //- do both point to same memory location
p != q; //- do they point to different locations
```

- **Limited arithmetic operations**

```
p = p + 1;    //- increments p by 4
d = d + 1;    //- increments d by 8
```

  - Will point to next item in an array

Pointer Arithmetic

- **Equivalence with Arrays:**

```
Array[3] ≡ *(Array + 3);
Array[3] ≡ Pointer[3];
```

- **Pointer Arithmetic:**

```
Pointer++          ++Pointer
Pointer--          --Pointer
Pointer += 5       Pointer -= 5
distance = Pointer1 – Pointer2
```

- **Pointer arithmetic works over pointer storage units**

```
int Array[5];
int* p0 = &Array[0];   //– p0 = 0x00010000
int* p1 = p0 + 1;      //– p3 = 0x00010004
int* p2 = &Array[1];   //– p1 = 0x00010004
```

Pointer Examples

```
int *pX = 5;          //– Warning: Memory Access
int X = 0;            //– X = 0
pX = &X;              //– OK
*pX = 5;              //– X = 5
*pX += 1;             //– X = 6
int Y = *pX + 1;      //– Y = 7
Y = *(pX + 1);        //– ERROR: out of bounds
int* pY = pX;         //– pY = &X
(*pY)++;              //– X = 7
*pY++;                //– ERROR: out of bounds
char* pA[2];          //– pA  is an array of ptrs to char
char (*pB)[2];        //– pB points to a char array
Printf("%d\n", *pX);
```

Initialisation of Pointer Arrays

- Only can initialise array of pointers to strings

```
char* Days(int n)
{
   static char *name[] = {"illegal",
          "Sunday", "Monday", "Tuesday",
              "Wednesday",Thursday",
"Friday",          "Saturday" };
    return ((n<1 || n>7)?name[0]:name[n];
}
```

- The `name` array is made `static` so that it is only created once when the process is loaded into memory.

Pointers and Function Arguments

- To pass a pointer as a value parameter (can't change where it points to) you just use *

```
void example(int* p)
{
   p = &x;     //- will be lost when it returns
}
```

- To pass a pointer as a reference parameter (can change where it points) you use * then &

```
void example(int* &p)
{
   *p = &x;    //- new value will be preserved
}
```

- A function can also return a pointer

```
int* example(…) {…}
```

Pointers and Function Arguments

WRONG	RIGHT

`Swap(int x, int y)`
`{`
`  int temp;`
`  temp = x;`
`  x = y;`
`  y = temp;`
`}`
	`Swap(int* pX, int* pY)`
	`{`
	`  int temp;`
	`  temp = *pX;`
	`  *pX = *pY;`
	`  *pY = temp;`
	`}`

- The function call would be

WRONG	RIGHT
`int *a, *b;` `Swap(a, b);`	`int a = 5, b = 3;` `Swap(&a, &b);`

Passing and Copying Arrays

- When arrays are passed as function arguments, only the base address is actually passed the array elements themselves are not copied.

```
//------ Using Indicies ----------//
strcpy(char to[], char from[])
{
  int i = 0;
  while ((to[i] = from[i]) != '\0') i++;
}
//------ Using Pointers ----------//
strcpy(char *to, char *from)
{
  while ((*to++ = *from++) != '\0');
}
```

const Type Qualifier

- Pointer to constant int, p can change but not *p

```
const int a;
const int *p = &a;          //- ptr to const int
int const *p = &a;          //- same as previous
```

- Constant pointer to int, *p can change but not p

```
int a;
int *const p = &a;          //- const ptr to int
```

- Here the value p points to can be modified, but the pointer itself must always point to the address of ‘a’.

Scope, Visibility & Storage classes

- The visibility of an identifier is determined by its scope
- The scope of an identifier is the part of the program in which the name can be used. This can be limited to the file, function, block, or function prototype where it appears.
- All identifiers except labels have their scope determined by the level at which the declaration occurs.
- The storage class of an identifier determines its lifetime.
- The static keyword can be used to modify the scope eg

  static int identifier;
- Global identifirs with the static storage-class specifier are visible only within the source file in which they are defined

File and Block

- File Scope:
  - **Identifier declarations that appear outside any block or list of parameters are accessible from anywhere in the file**
  - **These are known as “global” or “external” identifiers**
  - **The scope of a global begins at the point of its declaration and terminates at the end of the file.**
- Block Scope:
  - **Identifier declarations that appear inside a block or within a formal parameter list in a function definition.**
  - **Normally called local or automatic variables.**
  - **Only visible from the point of its declaration or definition to the end of the current block.**
  - **Its scope is limited to that block and to any nested blocks and ends at the curly brace that closes the current block.**

Storage Class and scope

Attributes		**Result**	
Item	Storage-class	Lifetime	Visibility
File Scope			
Variable defin.	Static	Global	Rest of file
Variable decl.	Extern	Global	Rest of file
Function proto. or definition	Static	Global	Single source file
Function proto.	extern	Global	Rest of file
Block Scope			
Variable decl.	extern	Global	Block
Variable defin.	static	Global	Block
Variable defin.	Automatic	Local	Block

Structures

- Example:

```
struct DATE {
     char day;          //- 1..31
     char month;        //- 1..12
     int  year;         //-
     int  dayofyear;    //- 1..352
}
```

- We instantiate it by the declaration :

```
struct DATE d;
```

- We initialise it by:

```
struct DATE d = {4, 7, 1766, 186};
```

- We reference it by :

```
d.day or d.month or d.year
```

Structures and Pointers

- If we declare

```
struct DATE *Date;
struct DATE *pDate = &Date;
```

- You can access a struct member as:

```
Date.year = 2013;
(*pDate).year = 2013;
```

- You can also do it using the '->' operator

```
pDate->year = 2013;
```

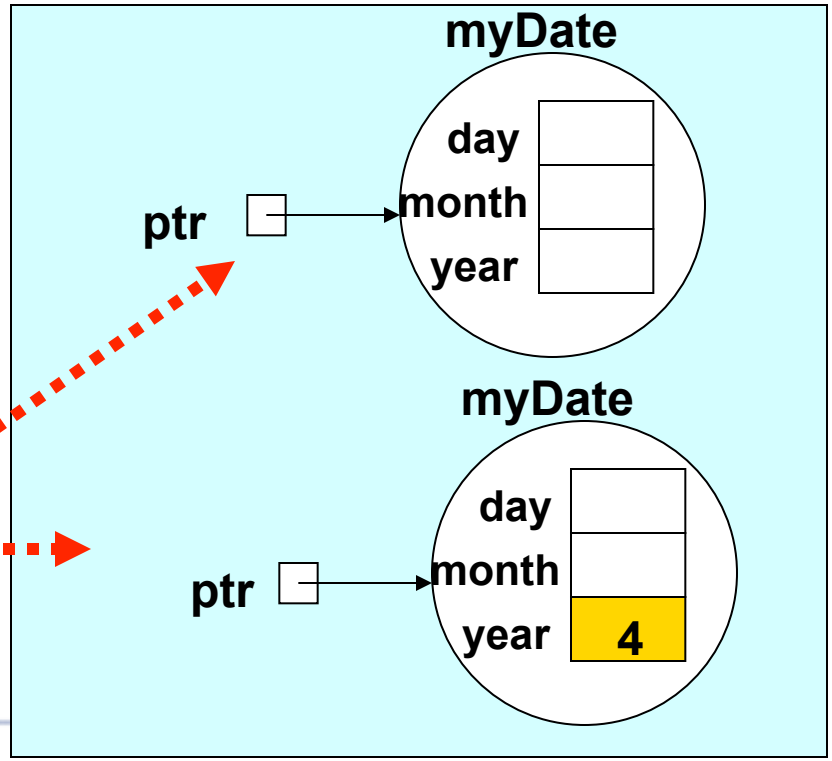
- The syntax for using the -> operator is:

```
pointerVariableName->classMemberName
```

Pointers to struct Example

```
struct Date
{
    char day;
    char month;
    int   year;
};

int main()
{
    struct Date *ptr;
    struct Date myDate;
    ptr = & myDate;
    ptr->day = 4;
    return 0;
}
```



Pointer Derefencing

```
struct Person {
    char          name[NAMESIZE];
    long          salary;
    struct DATE   birthdate;
    struct DATE*  hireddate;
} emplyee;
```

- If `struct Person* pP = &employee;`
- Then `pP->birthdate.year`
- Is the same as `(*pP).birthdate.year`
- Also `pP->birthdate->year`
- Is the same as `*(pP->hiredate).year`

Another Example

```
static int daysinMonth[2][13] = {
  {0,31,28,31,30,31,30,31,31,30,31,30,31 },
  {0,31,29,31,30,31,30,31,31,30,31,30,31 }
};
void set_date(stuct DATE* pD)
{
  int i = 1;
  int leap = pD->year % 4 == 0
          && pD->year % 100 != 0
          || pD-> year %400 ==0;
  pD->day = pD->dayofyear;
  while (pD->day > daysinMonth[leap][i])
  {
    pD->day -= daysinMonth[leap][i];
    pD->month = i++;
  }
}
```

Unions

- A union is a variable that can hold objects of different types and sizes, but only one at a time.
- The compiler keeps track of size and alignment.

```
union u_tag {
    int       ival;          //– either an int
    float     fval;          //– or a float
    char      name[8];       //– or a string
} uval;
```

- This can then be accessed by stating
- either `Union_name.member;`
- or `Union_pointer->member;`

Unions

- Unions are very useful in complex data structures

```
enum DTYPE {Integer, Floating, String};
struct array_element
{
   enum DTYPE type;
   union{int i; float f; char s[8]; };
};
struct array_element array[LENGTH];
```

- `Array` can now hold any combination of ints or floats or strings as specified in the `type` field

Static Arrays

- The size of static arrays is fixed at compile time
- Static arrays are easy to create, for example:

```
int array_name[2];
int array_name[3] = {3,2,1};
int array_name[] = {1,2,3,4};
int array_name[3][2][5];
```

- In C arrays are indexed using row-major order.
- This means the rightmost dimension changes fastest.

Strings

- A String is just a 1-D Array or characters
- The array must be NULL terminated
- Strings that are initialised at compile time are automatically null terminated as long as there is enough space in the array

```
char* name = “Jim”;    //- can not be modified
char name[] = “Jim”;   //- can be modified
char name[3] = “Jim”; //- no space for NULL
char name[4] = “Jim”; //- ok
```

- If not initialised by the compiler you have to manually make sure it is NULL terminated

String functions in C

- strcpy: Copy string
- strncpy: Copy n chars from string
- strcat: Concatenate/append strings
- strncat: Append n chars from string
- strcmp: Compare two strings
- strncmp: Compare characters of two strings
- strchr: Locate first occurrence of char in string
- strpbrk: Locate characters in string
- strrchr: Locate last occurrence of char in string
- strstr: Locate substring
- strtok: Split string into tokens
- strlen: Get string length

String functions in C

- #include <string.h>
- How long is a string?

```
char name[9] = “Jim”;
int size =strlen(“Jim”); //- size = 3; ignores NULL
int len = strlen(name);//- len = 3
```

- How to copy a string?

```
char name[9] = “Jim”;
char dest[9];              //- size must be >= strlen + 1
strcpy(dest, name);   //- copies NULL at end
strcpy(dest, “test”);    //- copies NULL at end
```

String functions in C

- Manual string copy

```
char src[9] = “Jim”;
char dest[9];
for (int i=0; i <= strlen(src); i++) dest[i] = src[i];
```

- How to compare strings?

```
char name[9] = “Jim”;
char label[5] = “Jan”;
if (strcmp(name, label) == 0)
      ; //- strings are the same
else ; //- strings are different
```

String functions in C

- How to find a substring in a string?

```
char str[9] = "JimJam";
char sub[5] = "Jan";
char* pos =  strstr(str, sub) ;
if (pos != NULL)  ; //- substring found
```

- How to find a char in a string?

```
char str[5] = "Jan";
int pos =  strchr(str, '.') ;
```

Escape Characters

- The escape sequences allow you to use a sequence of characters to represent special characters:

\a	Alert (bell)	\b	Backspace
\f	Formfeed	\n	Newline
\r	Carriage return	\\	Backslash
\t	Horizontal tab	\v	Vertical tab
\0	Null character		
\?	Literal quotation mark		
\'	Single quotation mark		
\"	Double quotation mark		
ddd	ASCII character in octal notation		
\x*hhh*	ASCII character in hex notation		

Multidimensional Arrays

- Multiple dimensional arrays can be created and accessed in a number of ways, if we have:

```
int array_name[ROW][COL];
int* p = array_name;
```

ROW = 2
COL = 5

- Then the following statements are all equivalent

```
int val = array_name[1][4];
int val = p[1 * COL + 4];
int val = *(p + 1 * COL + 4);
int val = *(array_name[1] + 4);
int val = (*(array_name + 1))[4];
```

- The reason for this is the way arrays are laid out.

Array[2][5]	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	0	1	2	3	4
Row 1	5	6	7	8	9

Initialising Arrays

- There are a number of ways to initialise multidim arrays

```
int m[2][3] = {1,2,3,4,5,6};
int m[2][3] = {{1,2,3},{4,5,6}};
int m[][3] = {{1,2,3}, {4,5,6}};
```

- If we properly delimit each row then we don't need to include zeros, these will be automatically initialised

```
int m[2][3] = {{1,2,0},{4,0,0}};
int m[2][3] = {{1,2},{4}};
```

- When the initialiser list is shorter than the number of array elements, the rest are set to zero, so we can initialise all the elements to zero by :

```
int array_name[100] = {0};
int m[2][3][3][4] = {0};
```

Multidimensional Arrays

- Another way to think of multidimensional arrays:
- *A 2D array is really a 1D array, each of whose elements is itself an array*
- This is why a[n][m] notation is used.
- Array elements are stored row by row.
- When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant.
- The reason for this is pointers. C needs to know how many columns so that it can jump from row to row in memory.
- Consider passing a[5][35] to a function:
- We can do: `f(int a[][35]) {.....}`
- or even: `f(int (*a)[35]) {.....}`
- We need parenthesis (*a) since [] have higher precedence than *

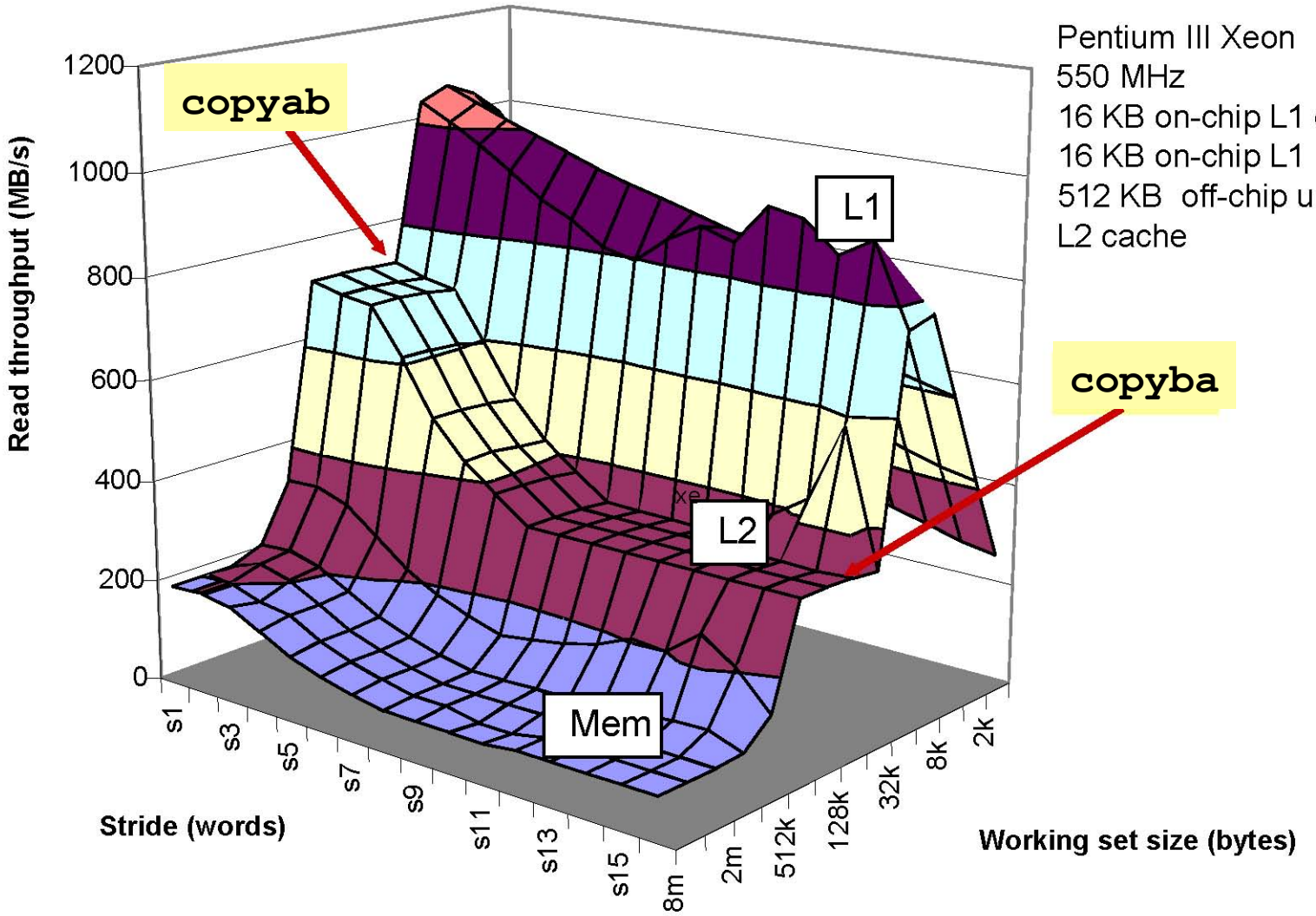
Array Performance

void copyab(int src[2048][2048], int dst[2048][2048])
{
for (int a = 0; a < 2048; a++)
for (int b = 0; b < 2048; b++)
dst[a][b] = src[a][b];
}
	void copyba(int src[2048][2048], int dst[2048][2048])
	{
	for (int b = 0; b < 2048; b++)
	for (int a = 0; a < 2048; a++)
	dst[a][b] = src[a][b];
	}

59,393,288 clock cycles	1,277,877,876 clock cycles

- (Measured on 2GHz Intel Pentium 4)
- 21.5 times slower!
- Performance depends on memory access patterns such as how you step through arrays

Memory Performance



The Stack

- The stack is an area of memory that is automatically setup when a program is loaded into memory
- The stack temporarily holds any arguments to functions and any variables that are defined local to the functions
- All these are called “automatic” variables because the compiler automatically allocates enough space for them on the stack, even if it is a large array or data structure.
- The stack also stores the return address of calling functions
- The compiler assigns a maximum size to the stack that can be set as a compiler option.
- Overflowing the stack limit will crash a program.
- Stack overflow can happen when using recursion or when creating an static array bigger than the stack size.

The Stack

- All variables declared locally within braces { } are created in an area of memory called the stack.
- These variables are deleted automatically when the closing brace is reached
- You can not return the address of an automatic variable
- But you can return its value

```
int* function()
{
    int number = 7;
    return &number; //- ERROR
}
```

- For large structures and objects or variables with lifetimes longer than a function it is better to use the system heap.

Stacks

- A stack is a list where
  - the addition and deletion of elements occur only at one end, called the top of the stack
  - *all elements are of the same type (WORDs),*
- Last In First Out (LIFO) data structure
  - Top element of stack is last element added to stack
  - Elements added (push) and removed (pop) from the top of the stack
  - Items added last are removed first
- Stacks are used to
  - implement function calls
  - convert recursive algorithms (especially not tail recursive) into nonrecursive algorithms

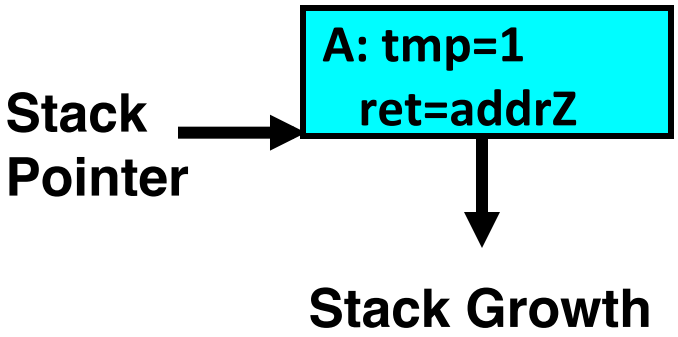
Execution Stack Example

```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
             C();
addrU:  }
        C()
        {
           A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```

- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

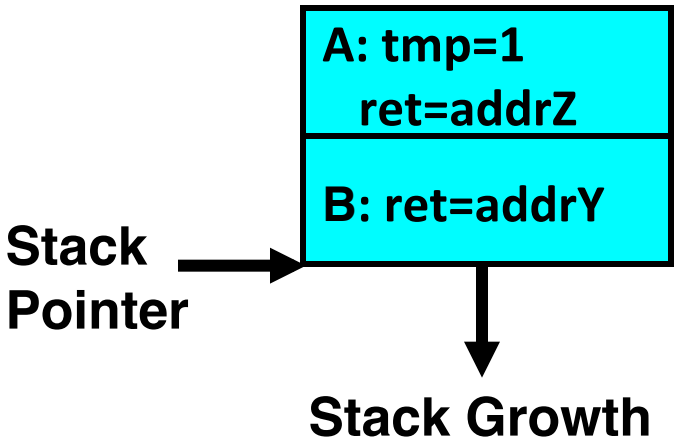
```
addrX:  A(int tmp)
        {
           if (tmp<2)
              B();
addrY:     printf(tmp);
        }
        B()
        {
           C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
           A(1);
addrZ:     exit;
        }
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

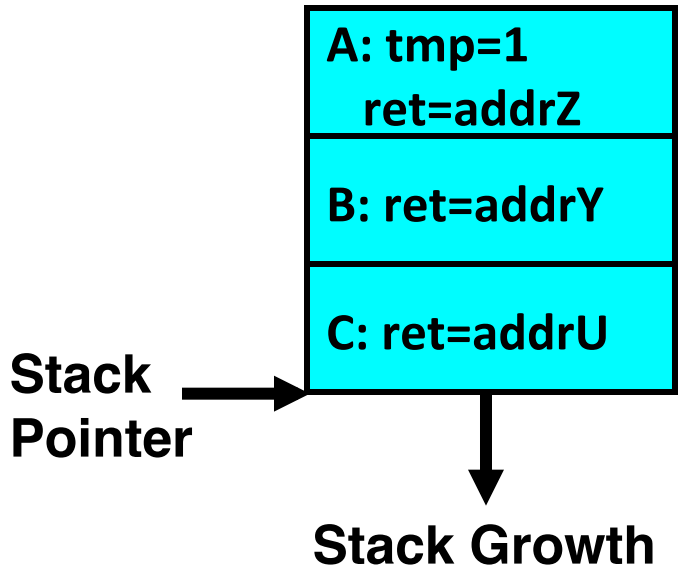
```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

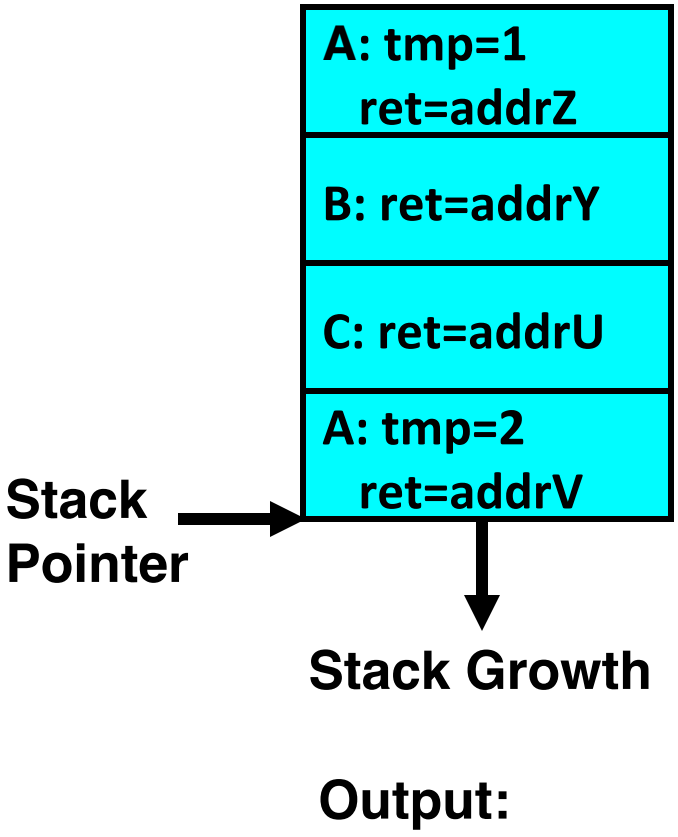
```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

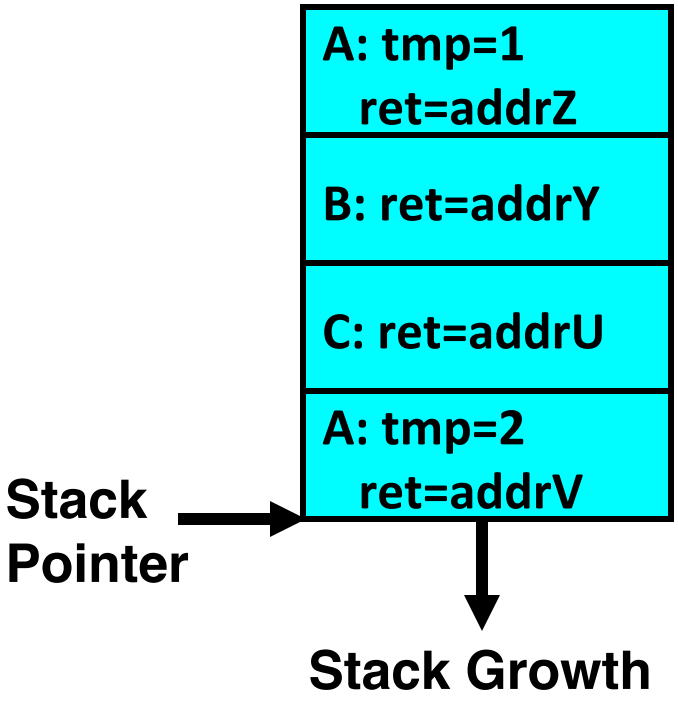
Execution Stack Example

```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```



Execution Stack Example

```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```

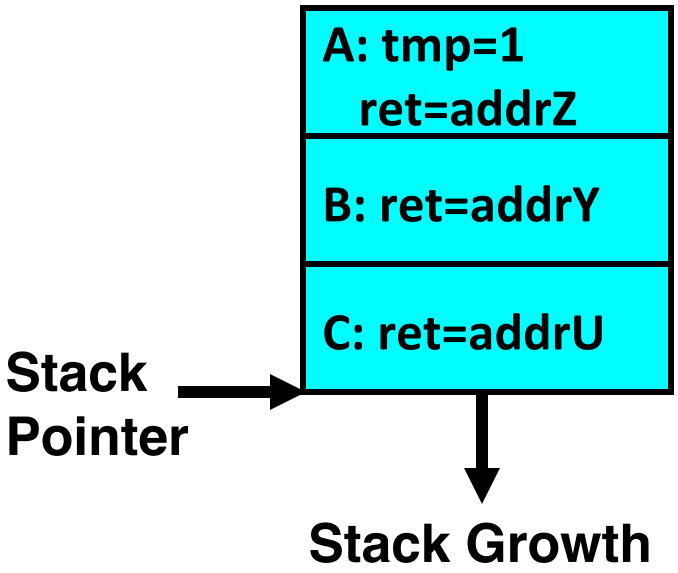


Output:

2

Execution Stack Example

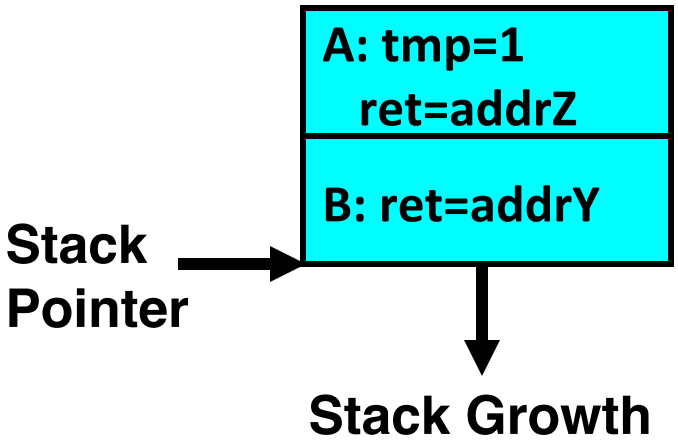
```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```



Output:
2

Execution Stack Example

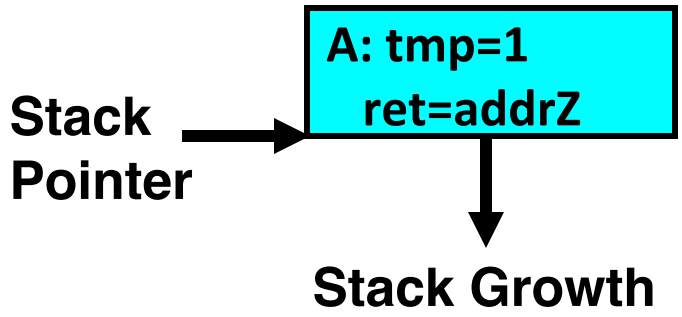
```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
           A(2);
addrV:  }
        void main()
        {
            A(1);
addrZ:      exit;
        }
```



Output:
2

Execution Stack Example

```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
            C();
addrU:  }
        C()
        {
          A(2);
addrV:  }
        void main()
        {
            A(1);
            exit;
addrZ:  }
```



Output:

2

1

Execution Stack Example

```
addrX:  A(int tmp)
        {
            if (tmp<2)
                B();
addrY:      printf(tmp);
        }
        B()
        {
             C();
addrU:  }
        C()
        {
           A(2);
addrV:  }
        void main()
        {
            A(1);
            exit;
addrZ:  }
```

Output:

2

1

Stack Memory Referencing Bug

```
main ()
{
  long int a[2];
  double d = 3.14;
  a[2] = 1073741824; /* Out of bounds reference */
  printf("d = %.15g\n", d);
  exit(0);
}
```

	Alpha	MIPS	Linux
-g	5.3049847741318e-315	3.1399998664856	3.14
-O	3.14	3.14	3.14

- (Linux version gives “correct” result, but implementing as separate function gives a segmentation fault.)

The Heap

- The **heap** (or free store) is a part of memory reserved by the OS for a program to use apart from the stack to store data whose size can not be determined at compile time.
- The size of the heap is limited only by the system's available virtual memory.
- At run-time the program requests as much free memory as it needs from the heap to store data.
- Unlike the stack frame, heap allocations must be freed manually. If heap memory is not freed the program will eventually run out of memory and will crash.
- A *memory leak* is caused when you forget to delete objects allocated on the heap.
- Creating space to store data on the heap at run time is called ***dynamic memory allocation***.

Dynamic Variables

- Most variables (automatic or static) are created at the start of a program.
- Dynamic variables must be created during execution of a program using pointers.
- Creating a dynamic variable allocates new memory on the heap to store data
- In C malloc() & free() functions are used.
- *In C++ as well as malloc() and free() you can also use `new` and `delete` to do memory dynamic management*

```
int* p3 = new int[5];    //- allocate it
delete [] p3;            //- deallocate it
p3 = NULL;               //- reset pointer
```

- *Never mix malloc/free with new/delete*

Dynamic Memory Allocation - C

- Requires *<stdlib.h>*
- To dynamically create space on the heap, use

  void *malloc(size_t ***size***); //- general use

  void *calloc(int ***num***, size_t ***size***);//- for arrays
  - ***calloc*() is specifically used for allocating arrays and is automatically initialised with all bits set to zero**
- Reallocating (resizing) memory blocks that have been created with *malloc*() is done using

  void *realloc(void ****memblock***, size_t ***size***);
- Any memory allocated using these functions must be released as soon as possible by using :

  void free(void ****memptr***);

  ***memptr* = NULL;**

Set any pointers to freed memory to NULL immediately, this will save you from a lot of debugging effort later!

Memory Management Functions

- **All dynamically allocated memory must be manually intialised before use. This can be done with the functions:**

  void *memset(void ***dest**, int ***c**, size_t ***count**);

  void *memcpy(void ***dst**, const void ***src**, size_t ***n**);

- **Other functions to manipulate the memory include :**

  void *memchr(const void ***buf**, int ***c**, size_t ***n**);

- **Returns pointer to first occurrence, within specified number of characters, of given character in buffer**

  int memcmp(const void ***p1**,const void ***p2**,size_t ***n**)

- **Compare the first *n* characters from two buffers.**

  void *memmove(void ***dest**,const void ***src**, size_t ***n**);

- **Copy the first *n* characters from one buffer to another**

New and Delete – C++

- **Syntax to use operator new**

  new dataType; //- allocate a single variable.

  new dataType[len]; //- allocate an array of vars

- **Syntax to use operator delete**

  delete pointer; //- destroy a dynamic variable

  delete [] pointer; //- destroy a dynamic array

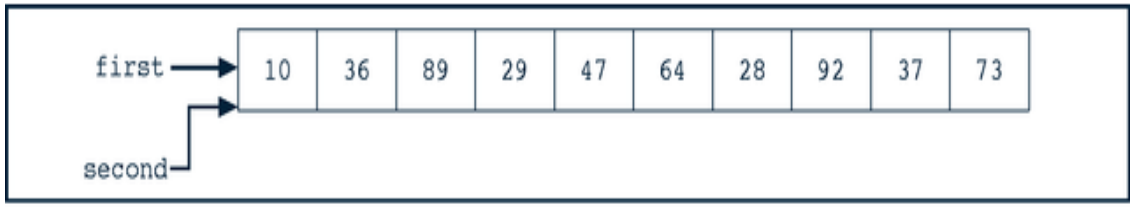
- **Examples**

```
int *ptr = new int;
char *str = new char[16];
*ptr = 5;
*str = "hello world";
delete ptr;
delete [] str;
```

Shallow Copy

- int * first = (int*)calloc(10, sizeof(int));

- int * second;
- second = first; //- shallow copy

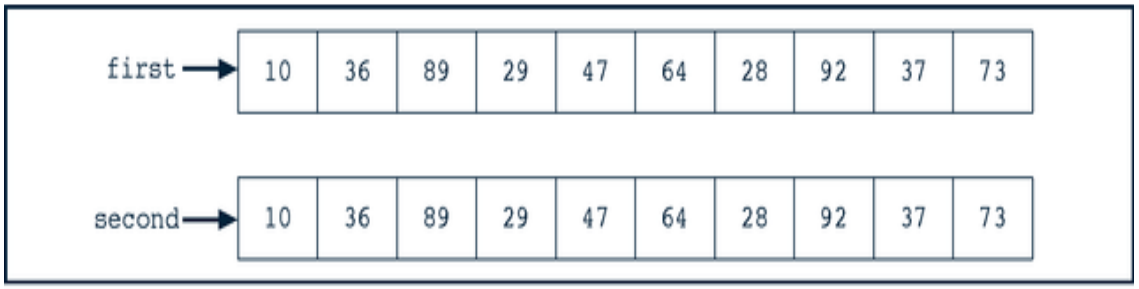


- free(second); //- removes array

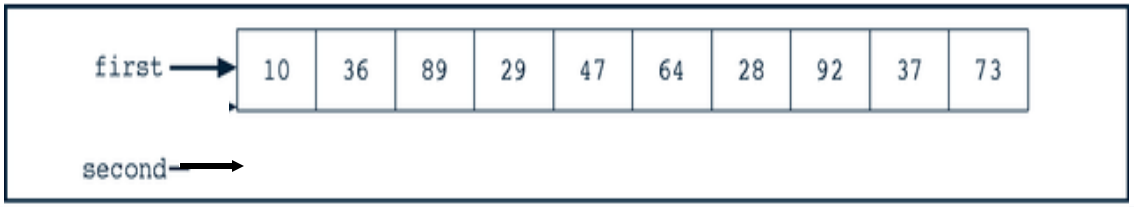
Deep Copy

- int * first = (int*)calloc(10, sizeof(int));
- int * second = (int*)calloc(10, sizeof(int));

```
for(int j = 0; j < 10; j++)
        second[j] = first[j];
```

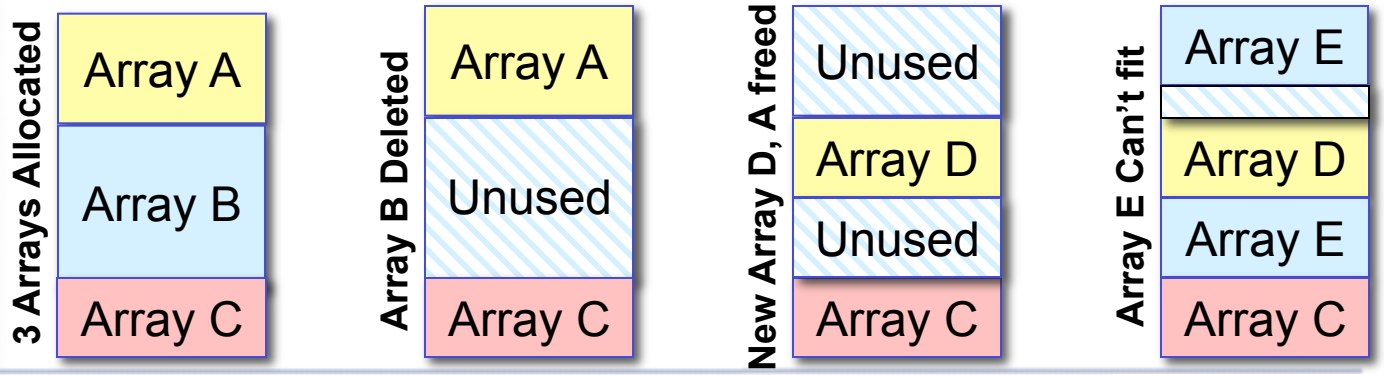


- free(second); //- removes array



Memory Allocation

- When a process allocates memory from the heap it uses up some of the heap's free space
- When the process frees memory it is deleted from the heap leaving a hole where it once was
- Any new allocation will be made in an unused area of memory but if it is too big to fit into any single block it mail fail to be be allocated
- Doing a lot of allocation/freeing can cause fragmentation and lead to running out of memory.



Ragged & Sparse Arrays – 1

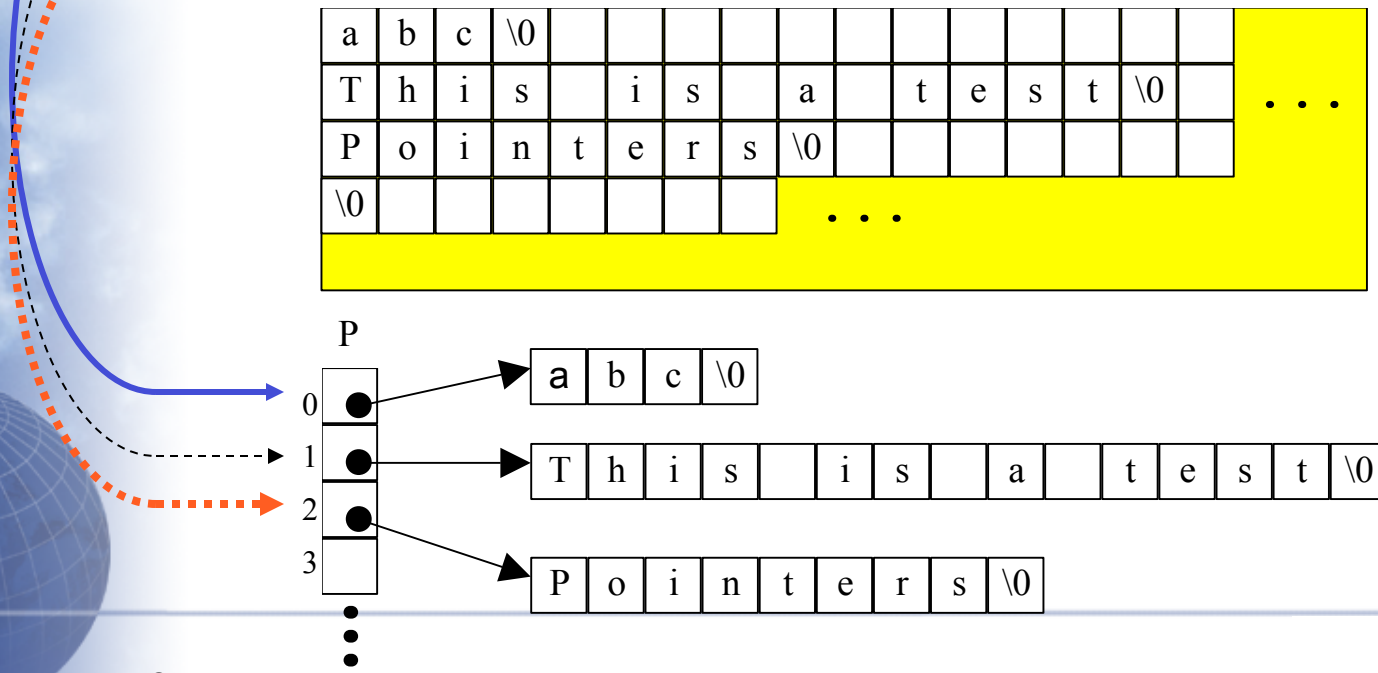
- One major advantage of using dynamic allocation is to conserve memory when dealing with arrays that are not fully utilised & avoid stack overflow
- For example you may wish to create an array of 50 strings which can vary from anywhere between 8 to 200 characters in length.
- Creating a two dimensional array is inefficient. In this case it is better to use a *ragged array*
- A ***ragged array*** is a one dimensional array of pointers to char (or some other type). Instead of

```
char strings[50][200] = {“abc”, “This is a test”};
char *pStrgs[50] = {“abc”, “This is a test”};
```

Ragged & Sparse Arrays – 2

- An ragged array can also be created as follows

```
char **p = (char**)calloc(50, sizeof(char*));
memset(pRag, NULL, 50*sizeof(char*));
p[0] = (char*)calloc(4, sizeof(char*));
p[1] = (char*)calloc(15, sizeof(char*));
p[n] = (char*)calloc(x, sizeof(char*));
```



Ragged & Sparse Arrays – 3

- This method lets us make 2D arrays of any size.

```
float **mat = calloc(nRows, sizeof(float*));
for (int x = 0; x < nRows; x++)
        mat[x] = calloc(nCols, sizeof(float));
```

- Elements in this array can be then accessed as

```
mat[i][j]= 3.0;
```

- To free the memory associated with this 2D array

```
for (x = 0; x < nRows; x++)
        free(mat[x]);
free(mat);
```


Windows Heaps

Programming Windows

- Can use one of three Application Programming Interfaces (API) to access OS functions (or a mix)
- Standard C Library
  - Highly portable
- Win32 API
  - Has its own set of conventions and programming techniques
  - Large and growing/Evolving
- POSIX API
  - “Portable Operating System Interface for uniX”
  - Formally designated IEEE 1003

Windows Principles

- Many system resources are represented as a *kernel object* identified and referenced by a *handle*
- Kernel Objects include files, processes, threads, pipes, memory mapping, events, etc.
- Kernel objects must be manipulated by Win32 API
  - Many functions perform the same or similar operations
  - Function calls will often have numerous parameters and flags, many of which can normally be ignored
  - Windows function names are long and descriptive
    - `WaitForSingleObject`
    - `WaitForMultipleObjects`
    - `WaitNamedPipe`

Windows Principles

- Predefined data types, required by the API, are in uppercase and descriptive
  - `BOOL`
  - `DWORD`
  - `LPSECURITY_ATTRIBUTES`
  - `HANDLE`
  - `LPTSTR`

  Many others..
- Predefined types avoid the * operator and make distinctions such as differentiating
  - `LPTSTR` (defined as `TCHAR *`)
  - `LPTCTSTR` (defined as const `TCHAR *`)
- Variable names in function prototype conventions
  - `lpszFileName` – Long pointer to zero-terminated string
  - `dwAccess` – double word (32 bits) containing file access flags

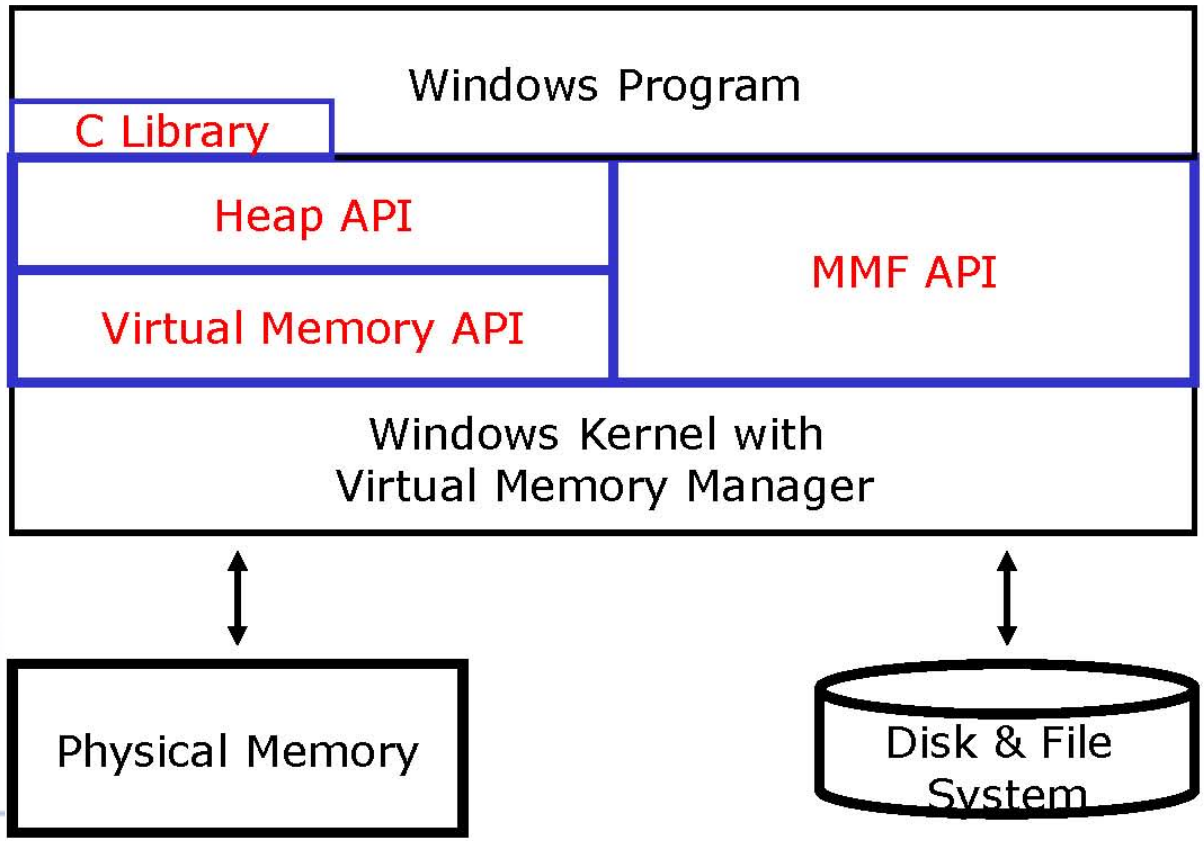
Hungarian Notation

- Parameter names are often long and descriptive (Hungarian Notation)

Character Datatype		Character Datatype	
c	char or count	p	pointer
b	byte (unsigned char)	fn	function
n	short or int	sz	string
b or f	BOOL (int)	l	LONG (long)
w	WORD (unsigned int)		
dw	DWORD (unsigned long)		
lph	long pointer to a handle		

Windows Heap API

- Most C programs use only a single heap, but Windows allows programs to use multiple heaps if they want to using the windows Heap API



Windows Heap Functions

- The main steps used with window heaps:
- 1. Get a heap handle using either `HeapCreate()` or `GetProcessHeap()`
- 2. Allocate blocks within the heap using `HeapAlloc()`
- 3. Optionally, free some or all of the individual blocks with `HeapFree()`
- 4. Destroy the heap and close the handle with `HeapDestroy()`

GetHeap and CreateHeap

```
HANDLE GetProcessHeap( VOID );
```

- Returns a handle to the default heap of the calling process used by malloc or NULL on failure

```
HANDLE HeapCreate(DWORD flOptions,
                  SIZE_T dwInitSize, SIZE_T dwMaxSize);
```

- Creates a new heap that can be used by the calling process and returns a handle for the heap or NULL on failure
- *dwMaxSize*
  - Reserve space in the virtual address space of the process
- *dwInitSize*
  - Allocate physical storage for a specified initial portion
- *flOptions*
  - HEAP_GENERATE_EXCEPTIONS
  - HEAP_NO_SERIALIZE

GetHeap and CreateHeap

```
BOOL HeapDestroy( HANDLE hHeap);
```

- A function used for destroying an entire heap. Destroying a heap is a quick way to free date structures without traversing them to delete one element at a time. **Be careful not to destroy the process's heap.**

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags,
                 SIZE_TdwBytes);
```

- **Return - a pointer to the allocated memory, or NULL on failure**
- ***dwFlags***
  - **HEAP_GENERATE_EXCEPTIONS**
  - **HEAP_NO_SERIALIZE**
  - **HEAP_ZERO_MEMORY**

```
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags,
              LPVOID lpMem);
```

- Frees a memory block in *hHeap* pointed to by *lpMem*
- Returns 0 on failure, otherwise nonzero

CreateHeap Example

```
SIZE_T nBufferSize  = 0xFFFFFF
HANDLE hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS
                          | HEAP_NO_SERIALIZE, nBufferSize, 0);
if (hHeap == NULL) exit(-1);
 LPVOID ptrHeap = HeapAlloc(hHeap,
                           HEAP_ZERO_MEMORY,
                           sizeof(TCHAR) * nBufferSize);
if (ptrHeap == NULL)
{
      HeapDestroy(hHeap);
      exit(-1);
}
```

HeapRealloc and HeapSize

```
LPVOID HeapReAlloc(HANDLE hHeap,
        DWORD dwFlags,
        LPVOID lpMem, SIZE_T dwBytes);
```

- Reallocates (resizes) a block of memory from a heap pointed to by ***lpMem*** to the new size of ***dwBytes***
- **Return:** A pointer to the reallocated memory block, or NULL on failure

```
DWORD HeapSize(HANDLE hHeap, DWORD dwFlags,
                LPCVOID lpMem);
```

- returns the size of the back pointed to be *lpMem* allocated from a heap using either HeapAlloc or HeapRealloc

Questions?