C++ Data Structures
Prerequisites

CHAPTER 1: C++ POINTERS/REFERENCES

DR. ERIC CHOU

IEEE SENIOR MEMBER

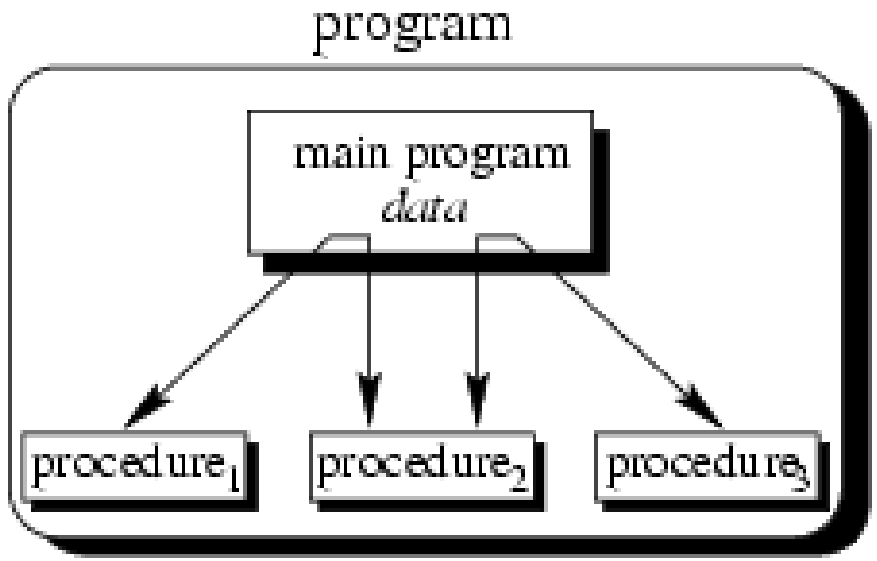
Objectives

- Understand what is pointer and what is reference in C++ language.
- C++ Pointers, References (alias), Objects, Structures, Containers

LECTURE 1

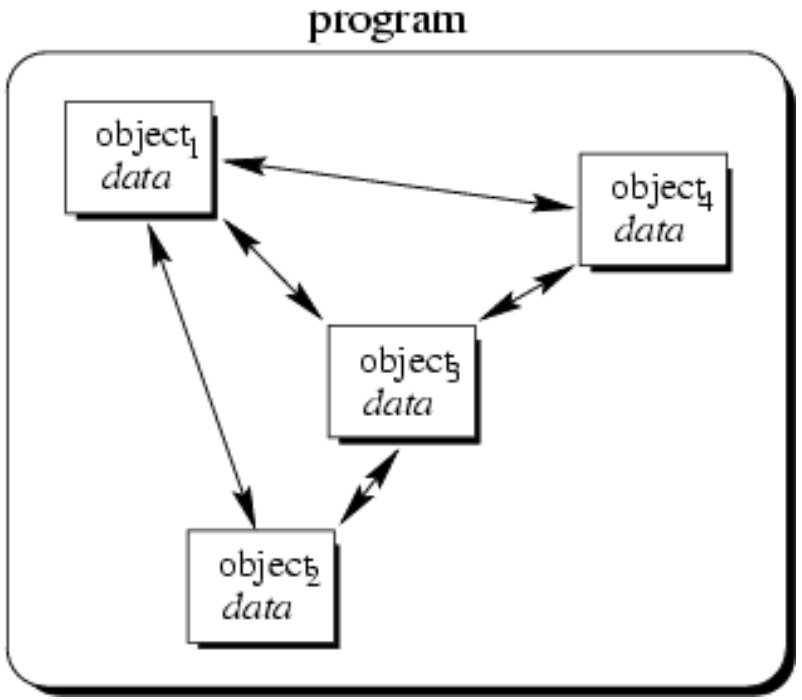
C++ Data Abstraction

Procedural Concept



- The main program coordinates calls to procedures and hands over appropriate data as parameters.

Object-Oriented Concept



•Objects of the program interact by sending messages to each other

C++

- Supports **Data Abstraction**
- Supports **OOP**
  - Encapsulation
  - Inheritance
  - Polymorphism
- Supports **Generic Programming**
  - Containers
    - Stack of char, int, double etc
  - Generic Algorithms
    - sort(), copy(), search() any container Stack/Vector/List

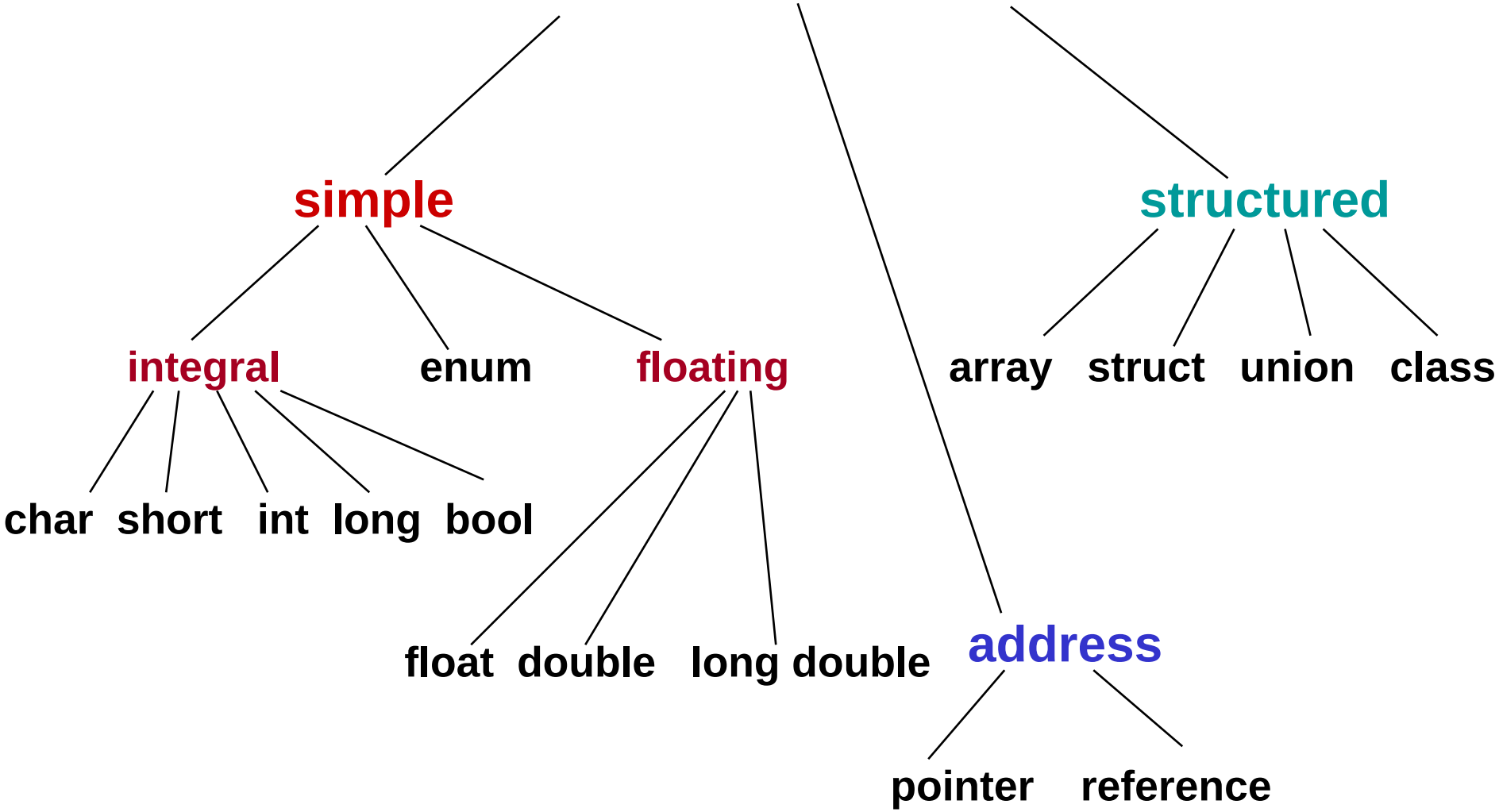
LECTURE 2

C++ Data Abstraction Mechanisms

Pointers, Dynamic Data, and Reference Types

- Review on Pointers
- Reference Variables
- Dynamic Memory Allocation
  - The new operator
  - The delete operator
  - Dynamic Memory Allocation for Arrays

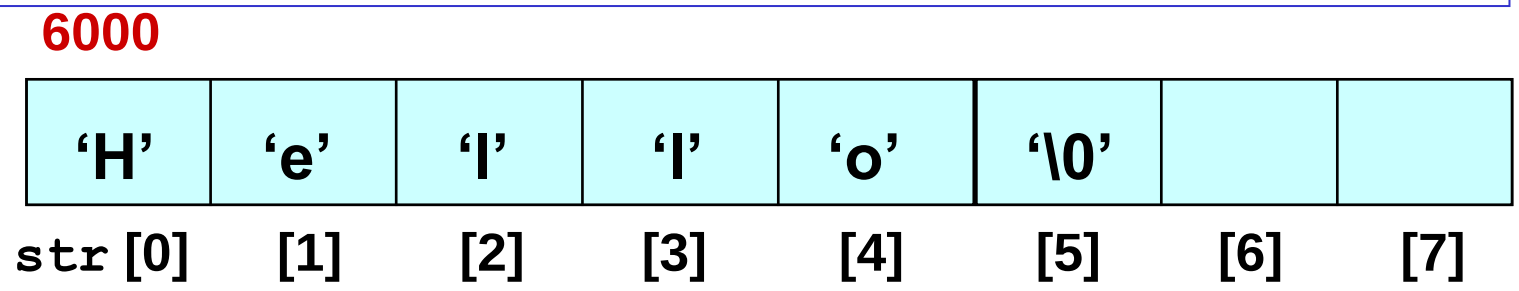
C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

Recall that . . .

char str [8];

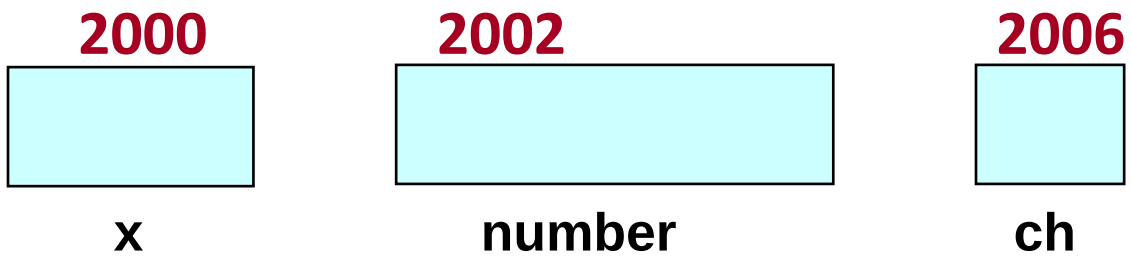
- **`str`** is the base address of the array.
- We say **`str`** is a pointer because its value is an address.
- It is a pointer constant because the value of **`str`** itself cannot be changed by assignment. It "points" to the memory location of a `char`.



Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

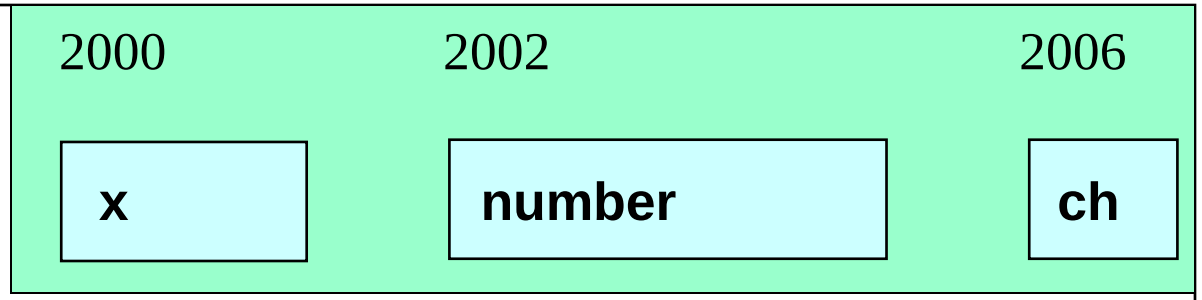
```
int     x;
float   number;
char    ch;
```



Obtaining Memory Addresses

- The address of a *non-array variable* can be obtained by using the address-of operator &

```
int     x;
float   number;
char    ch;
```




```
cout << "Address of x is " << &x << endl;

cout << "Address of number is " << &number << endl;

cout << "Address of ch is " << &ch << endl;
```

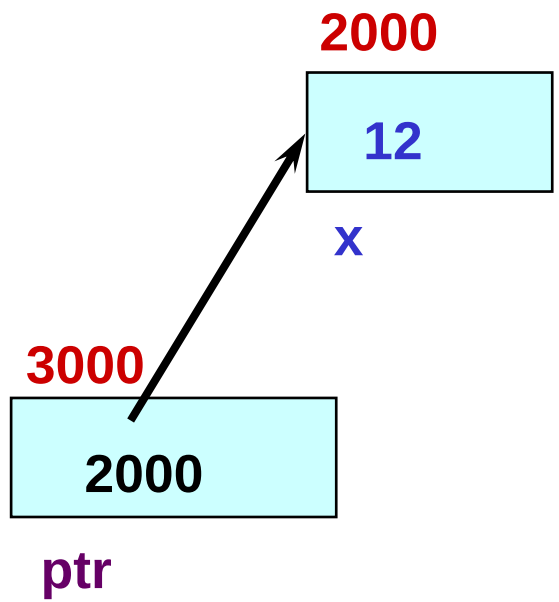
What is a pointer variable?

- A pointer variable is a variable whose value is the address of a location in memory.
- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

```
int*   ptr; // ptr will hold the address of an int

char*  q;   // q will hold the address of a char
```

Using a Pointer Variable

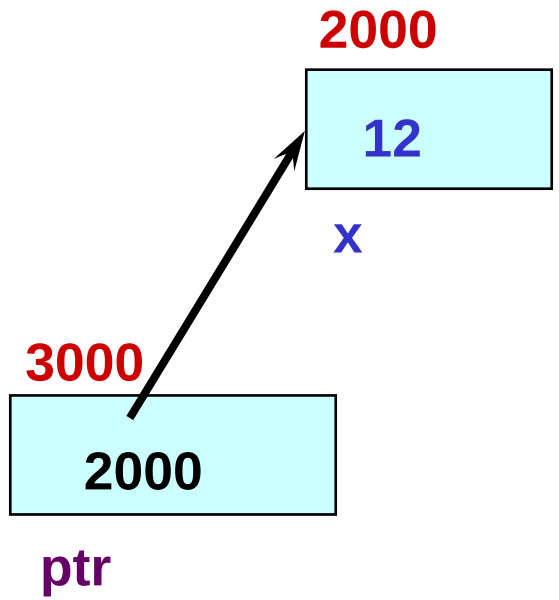
```
int  x;
x = 12;

int*  ptr;
ptr = &x;
```



NOTE: **Because ptr holds the address of x, we say that ptr "points to" x**

*: dereference operator

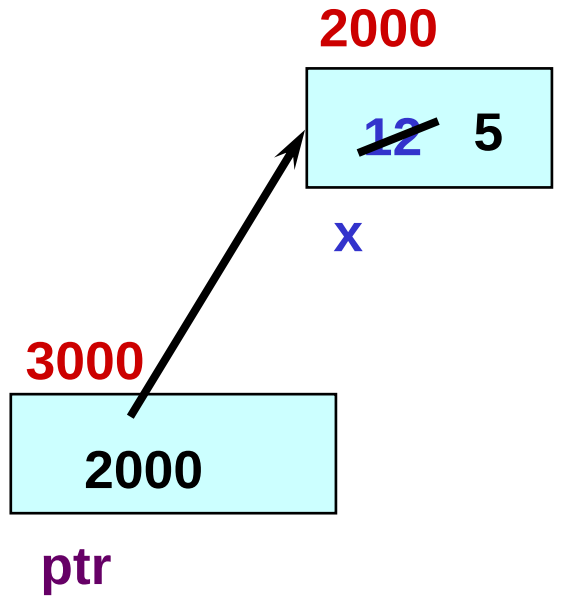
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



NOTE: The value pointed to by ptr is denoted by *ptr

Using the Dereference Operator

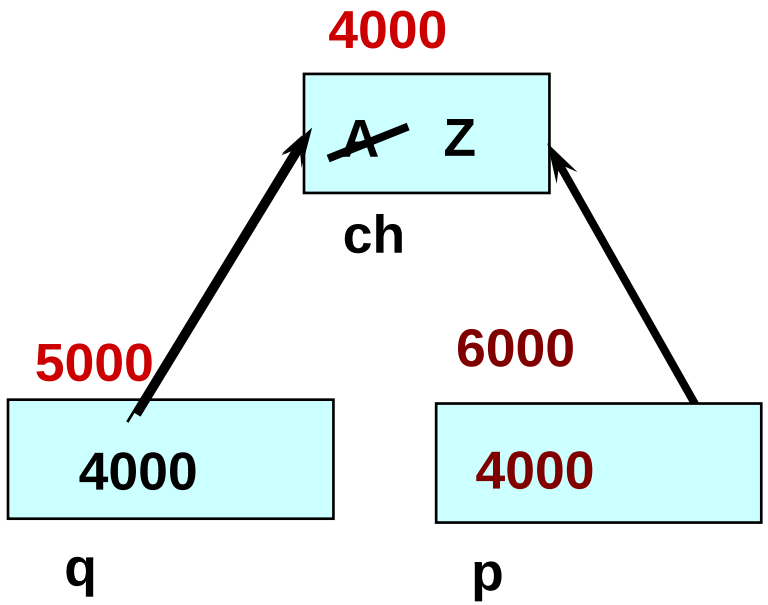
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;
```




```
// changes the value at the
   address ptr points to 5
```

Self –Test on Pointers

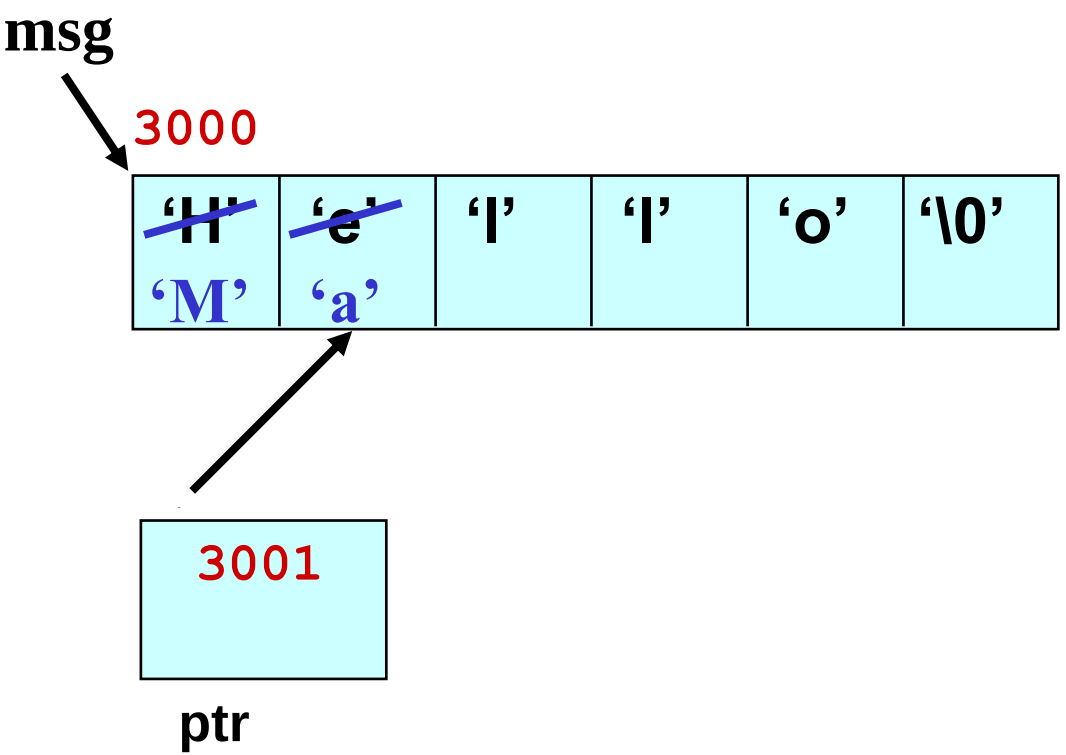
```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;
```




```
// the rhs has value 4000
// now p and q both point to ch
```

Using a Pointer to Access the Elements of a String

```
char   msg[ ] =“Hello”;
char*  ptr;
ptr  =  msg;
*ptr  = ‘M’ ;
ptr++;
*ptr = ‘a’;
```



Reference Variables

Reference variable = ***alias for another variable***

- Contains the address of a variable (like a pointer)
- No need to perform any dereferencing (unlike a pointer)
- Must be initialized when it is declared

```
int x = 5;
int &z = x;            // z is another name for x
int &y ;               //Error: reference must be initialized
cout << x << endl;     -> prints 5
cout << z << endl;     -> prints 5

z = 9;                 // same as x = 9;
cout << x << endl;     -> prints 9
cout << z << endl;     -> prints 9
```

Why Reference Variables

- Are primarily used as function parameters

- **Advantages of using references:**
  - you don’t have to pass the address of a variable
  - you don’t have to dereference the variable inside the called function

Reference Variables Example

```
#include <iostream.h>
// Function prototypes
  (required in C++)

void p_swap(int *, int *);
void r_swap(int&, int&);

int main (void){
 int v = 5, x = 10;
 cout << v << x << endl;
 p_swap(&v,&x);
 cout << v << x << endl;
 r_swap(v,x);
 cout << v << x << endl;
 return 0;
}
```

```
void p_swap(int *a, int *b)
{
  int temp;
   temp = *a;      (2)
   *a = *b;        (3)
   *b = temp;

}
```

```
void r_swap(int &a, int &b)
{
 int temp;
 temp = a;        (2)
 a = b;           (3)
 b = temp;
}
```

Dynamic Memory Allocation

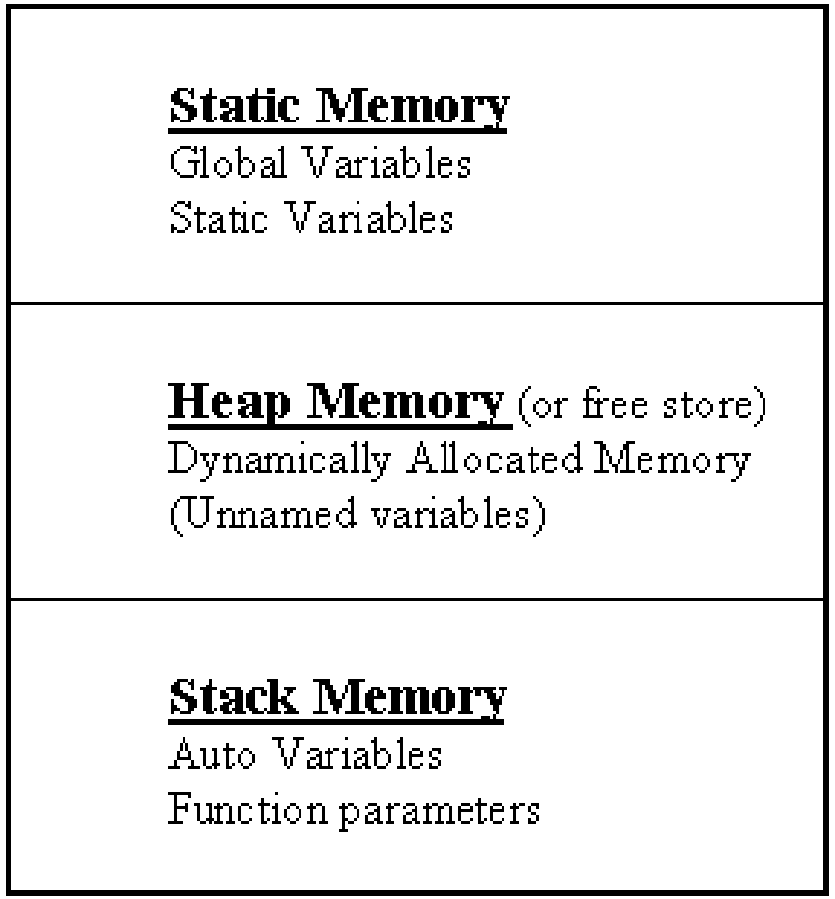
In C and C++, three types of memory are used by programs:

Static memory - where global and static variables live

Heap memory - dynamically allocated at execution time

- "managed" memory accessed using pointers

Stack memory - used by automatic variables



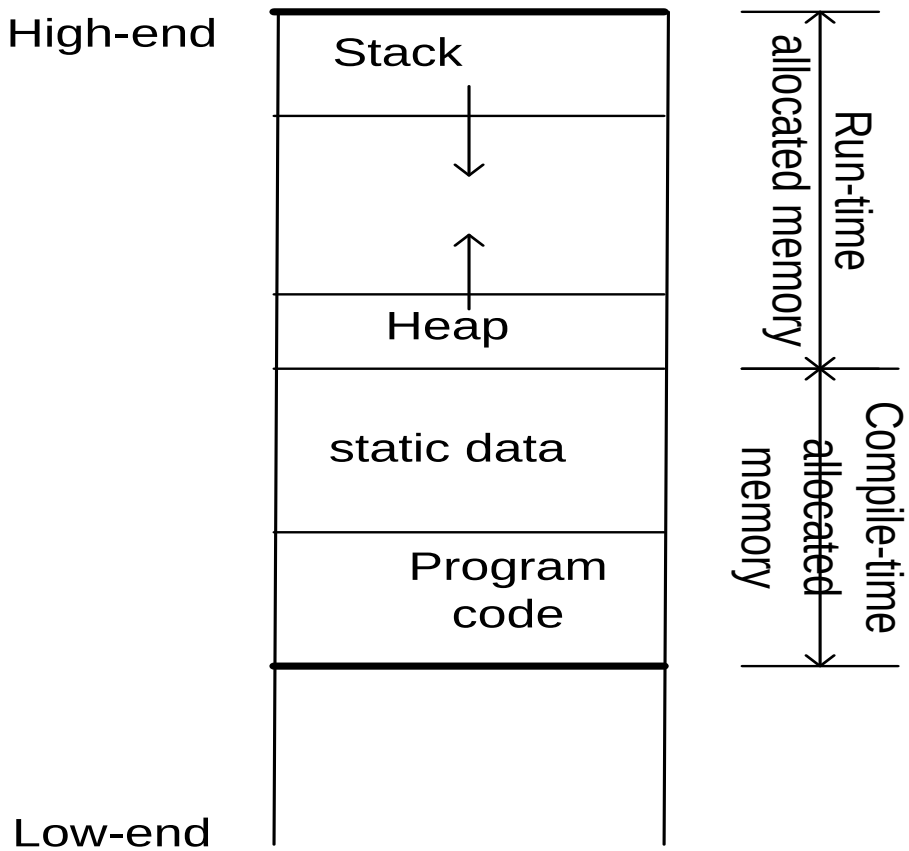
3 Kinds of Program Data

- **STATIC DATA**: Allocated at compiler time

- **DYNAMIC DATA**: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators **new** and **delete**

- **AUTOMATIC DATA**: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function

Dynamic Memory Allocation Diagram



Dynamic Memory Allocation

- In C, functions such as malloc() are used to dynamically allocate memory from the **Heap**.
- In C++, this is accomplished using the **new** and **delete** operators
- **new** is used to allocate memory during execution time
  - returns a pointer to the address where the object is to be stored
  - always returns a pointer to the type that follows the **new**

Operator new Syntax

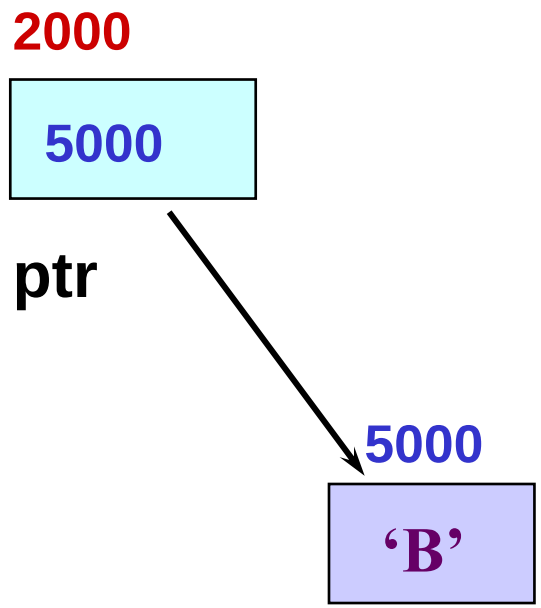
```
new   DataType
```

```
new   DataType  [IntExpression]
```

- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- Otherwise, program terminates with error message.
- The dynamically allocated object exists until the delete operator destroys it.

Operator new

```
char*  ptr;

ptr = new char;
*ptr = 'B';
cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

The **NULL** Pointer

- There is a pointer constant called the “null pointer” denoted by NULL
- But NULL is not memory address 0.
- NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer’s job to check for this.

```
while (ptr != NULL) {
        . . .              // ok to use *ptr here
}
```

Operator **delete** Syntax

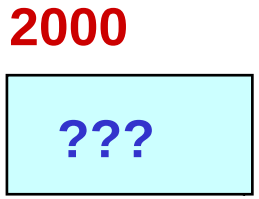
```
delete    Pointer
```

```
delete [ ]    Pointer
```

- The object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.
- Good idea to set the pointer to the released memory to NULL
- Square brackets are used with delete to deallocate a dynamically allocated array.

Operator **delete**

```
char*  ptr;



ptr = new char;

*ptr = 'B';

cout  <<  *ptr;

delete  ptr;
```

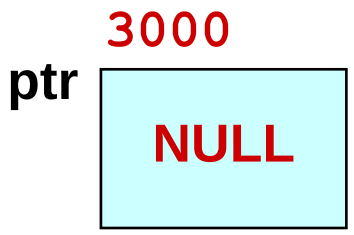


NOTE:

delete deallocates the memory pointed to by ptr

Example

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 0 ] = ‘u’;


delete [] ptr;

ptr = NULL;
```



// deallocates the array pointed to by ptr
// ptr itself is not deallocated
// the value of ptr becomes undefined

Pointers and Constants

```
char* p;
p = new char[20];

char c[] = "Hello";
const char* pc = c; //pointer to a constant
pc[2] = 'a'; // error
pc = p;

char  *const cp = c; //constant pointer
cp[2] = 'a';
cp = p; // error

const char *const cpc = c; //constant pointer to a const
cpc[2] = 'a'; //error
cpc = p; //error
```

LECTURE 3

Pointers and References

Pointers and References

- C and C++ support pointers which are different from most of the other programming languages. Other languages including C++, Java, Python, Ruby, Perl and PHP support references.
- On the surface, both references and pointers are very similar, both are used to have one variable provide access to another. With both providing lots of the same capabilities, it's often unclear what is different between these different mechanisms. In this article, I will try to illustrate the differences between pointers and references.

Pointers and References

- Pointers: A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.
- References : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.
  A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the * operator for you.

Pointers and References

```
int i = 3;


// A pointer to variable i (or stores
// address of i)
int *ptr = &i;


// A reference (or alias) for i.
int &ref = i;
```

LECTURE 3

Comparison Between Pointers and References

1. Initialization:
A pointer can be initialized in this way:

```
int a = 10;
 int *p = &a;
        OR
   int *p;
  p = &a;
```

we can declare and initialize pointer at same step or in multiple line.

2. Initialization:
While in references,

```
int a=10;
int &p=a;  //it is correct
   but
int &p;
 p=a;     // it is incorrect as we should declare and
          // initialize references at single step.
```

3. NOTE:

- This differences may vary from compiler to compiler.
- The above differences is with respect to turbo IDE.

4. Reassignment:

- A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. See the following examples:

```
int a = 5;
int b = 6;
int *p;
p =  &a;
p = &b;
```

5. Reassignment:

- On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

6. Memory Address:

- A pointer has its own memory address and size on the stack whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack.

```
int &p = a;
cout << &p << endl << &a;
```

7. NULL value:

- Pointer can be assigned NULL directly, whereas reference cannot. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into exception situation.

8. Indirection:

- You can have pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.I.e,

In Pointers,

```
int a = 10;
int *p;
int **q;  //it is valid.
p = &a;
q = &p;
```

Whereas in references,

```
int &p = a;
int &&q = p; //it is reference to
reference, so it is an error.
```

9. Arithmetic operations:

- Various arithmetic operations can be performed on pointers whereas there is **no** such thing called **Reference Arithmetic**.
- but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

LECTURE 4

When to use What?

When to use what?

- The performances are exactly the same, as references are implemented internally as pointers. But still you can keep some points in your mind to decide when to use what :

- Use references
  - In function **parameters** and return **types**.

- Use pointers:
  - Use pointers if pointer arithmetic or passing NULL-pointer is needed. For example for arrays (Note that array access is implemented using pointer arithmetic).
  - To implement data structures like linked list, tree, etc and their algorithms because to point different cell, we have to use the concept of pointers.