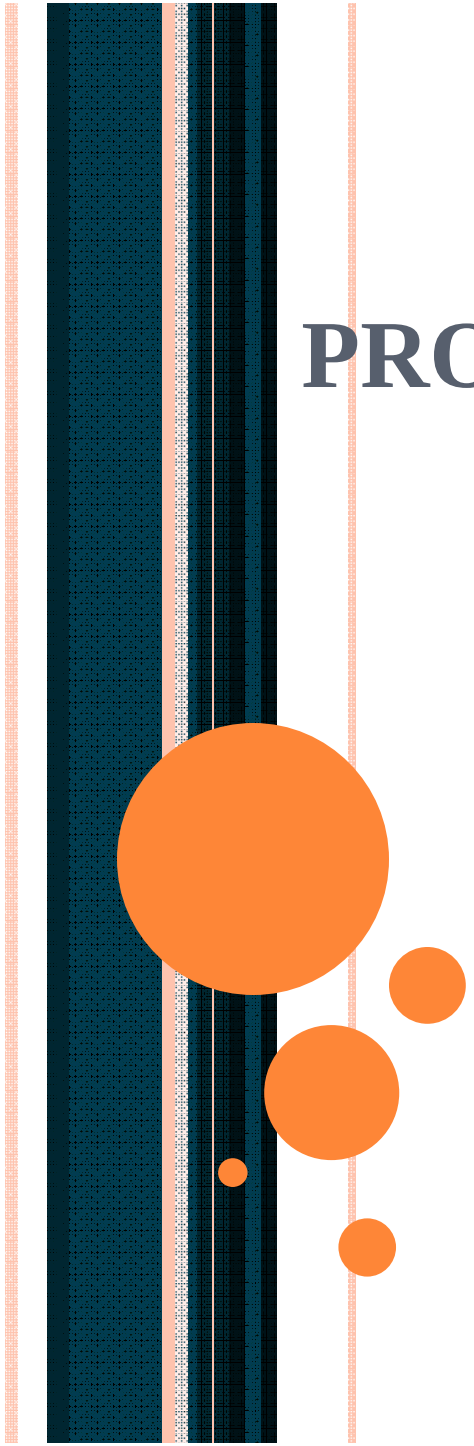
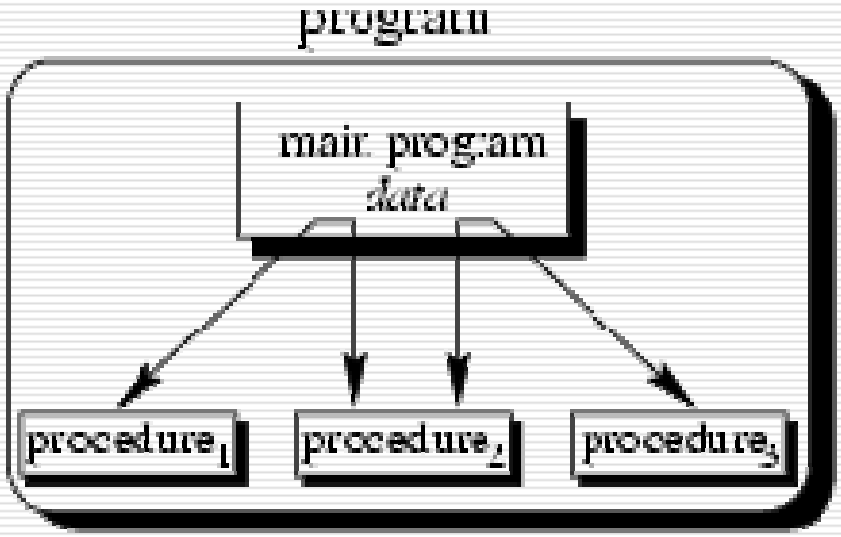


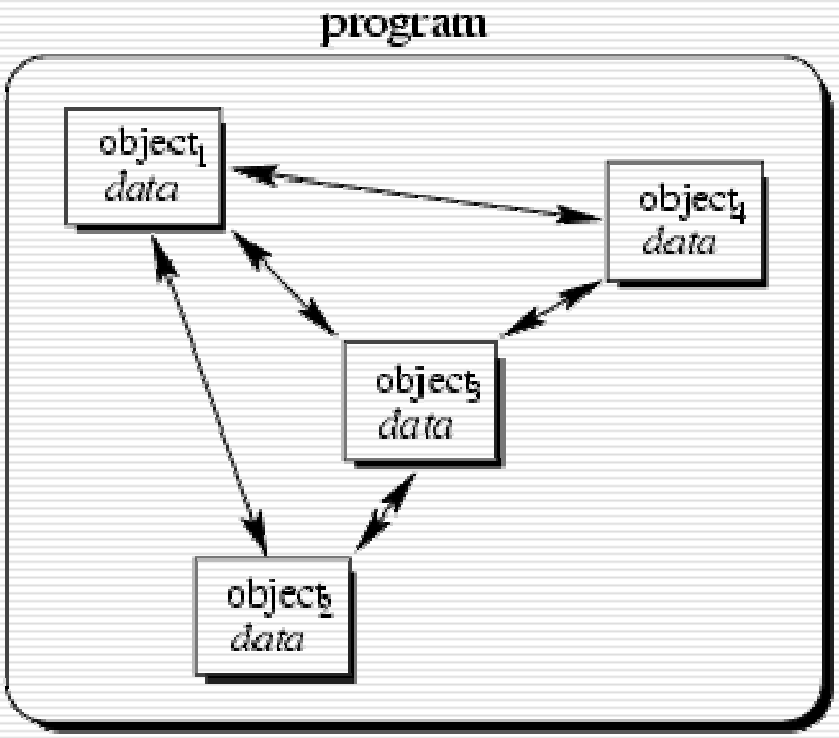
OBJECT ORIENTED PROGRAMMING USING C++

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

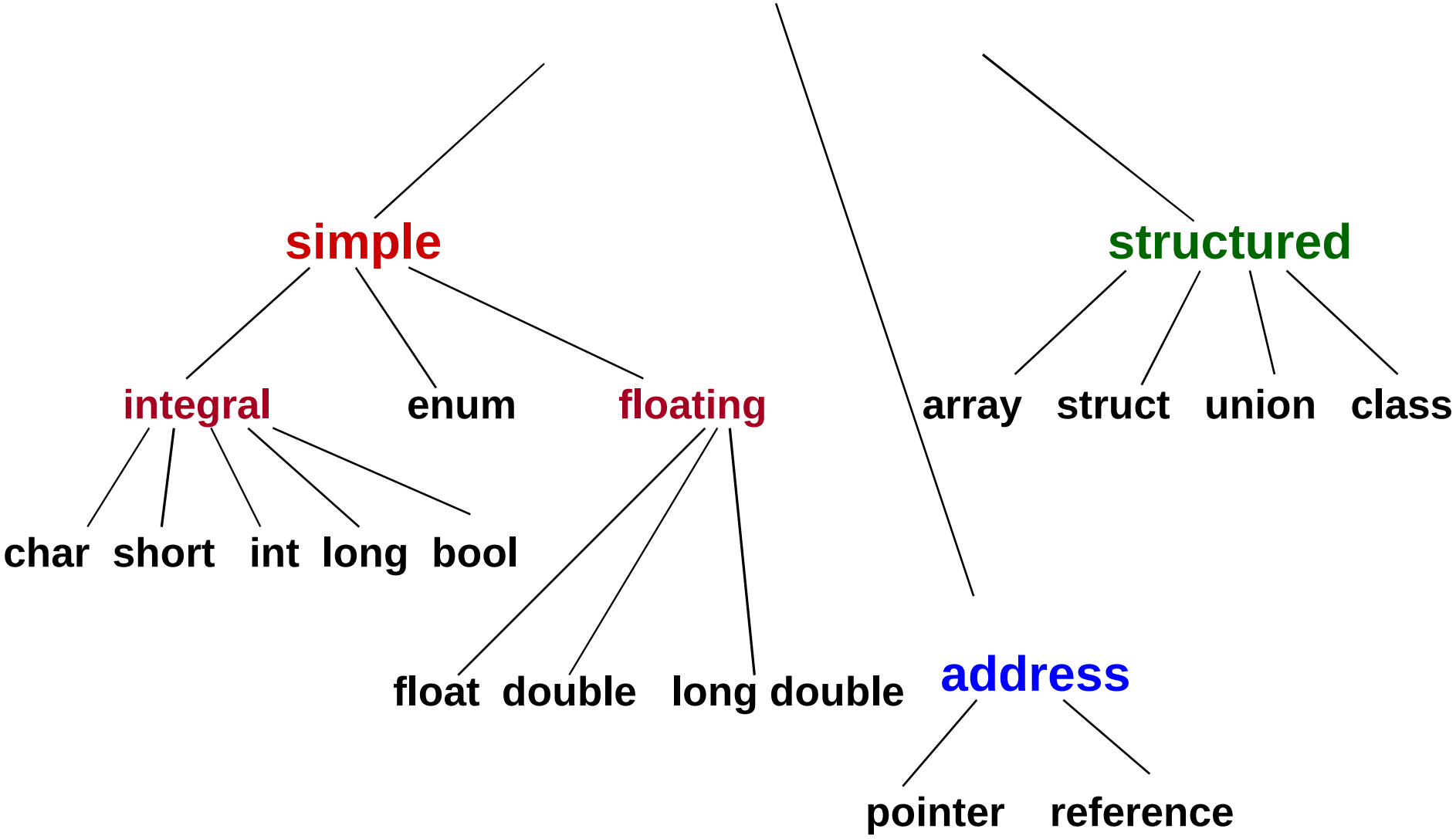
C++

- 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*

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

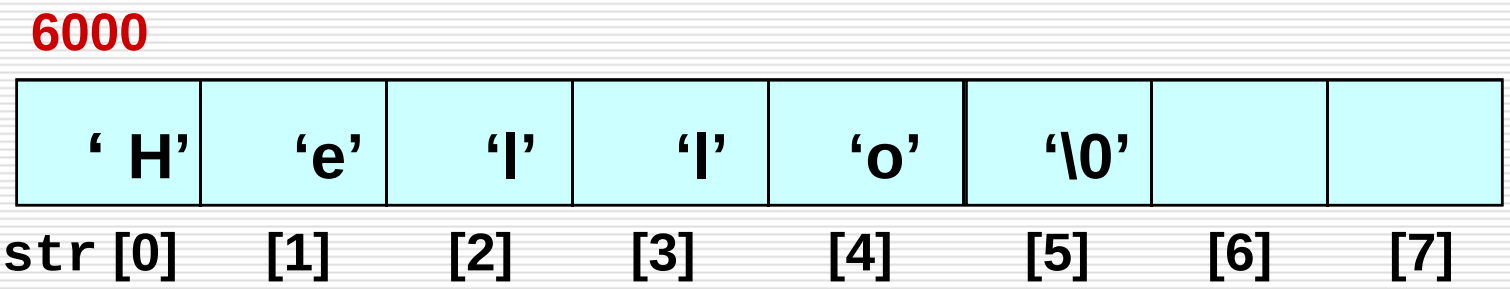
C++ Data Types


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

Array Basics

`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.



String Literals

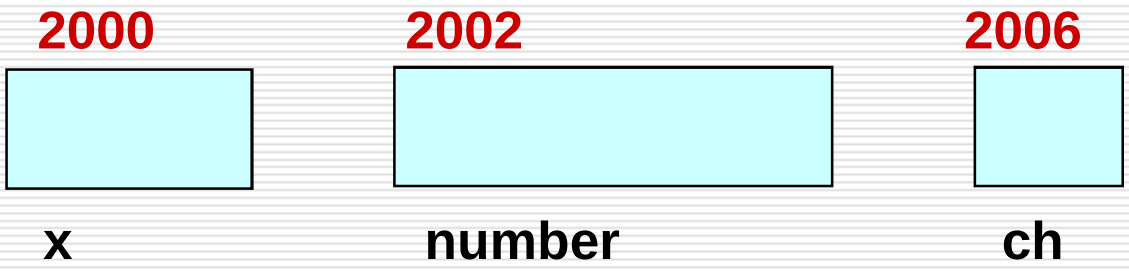
```
char* p = “Hello”;
char[] q = “Hello”;
char* r = “Hello”;
```

- p[4] = ‘O’; // error: assignment to constant
- q[4] = ‘O’; // ok, q is an array of 5 characters
- p == r; // false; implementation dependent

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

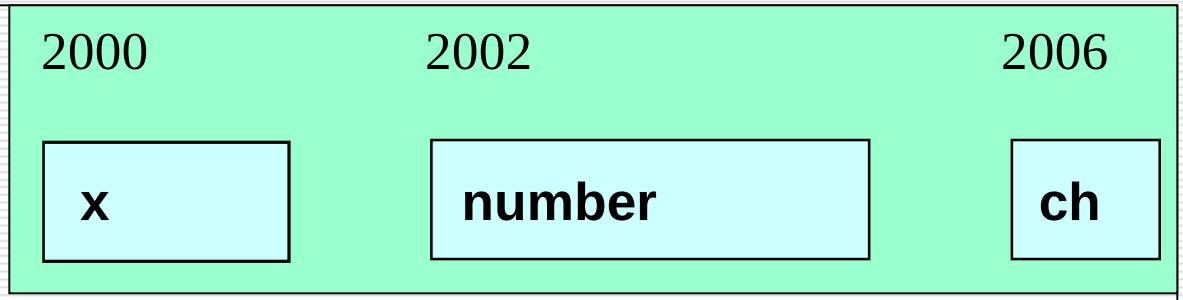
```
int    x;
float  number;
char   ch;
```



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;
```

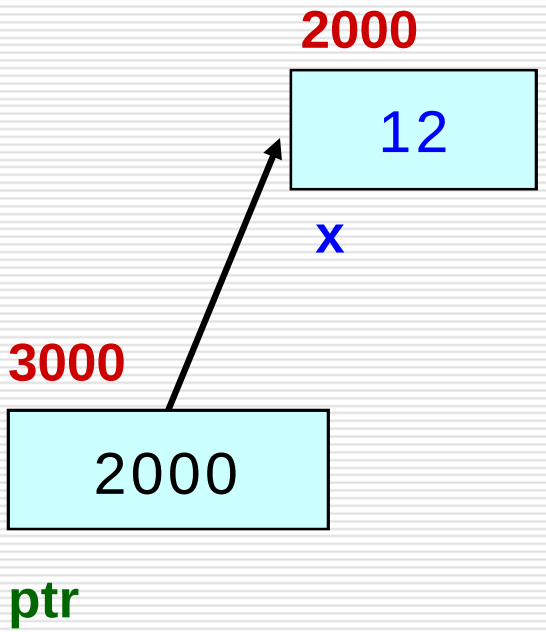
What is a Pointer Variable

- 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

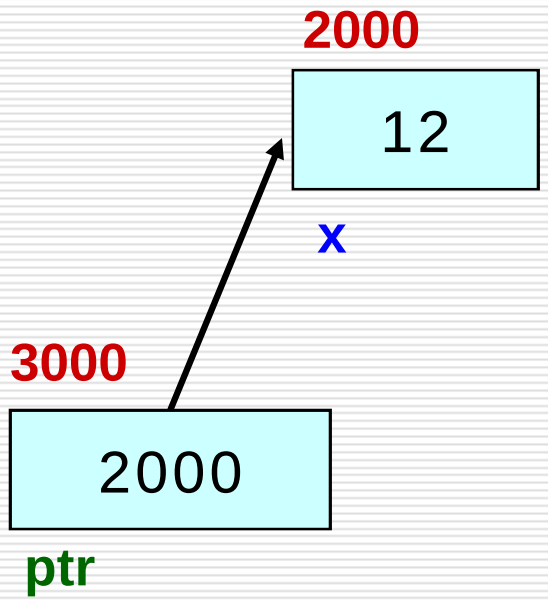
```
int  x;
x = 12;

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



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

*: Dereference Operator

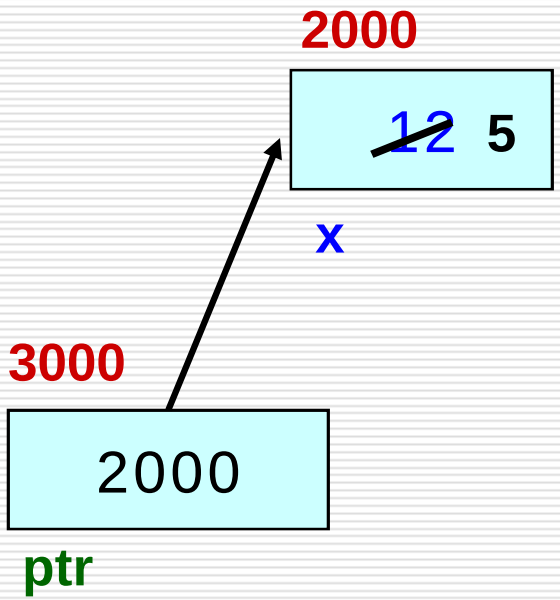
```
int  x;
x = 12;

int* ptr;
ptr = &x;

cout << *ptr;
```



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

Using the Dereference Operator

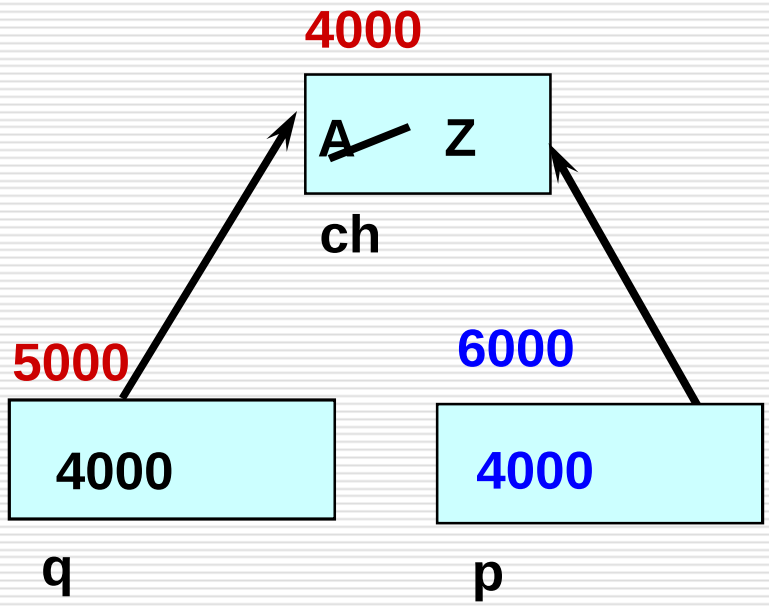
```
int  x;
x = 12;

int*  ptr;
ptr = &x;

*ptr = 5;
```



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

Self-Test on Pointers

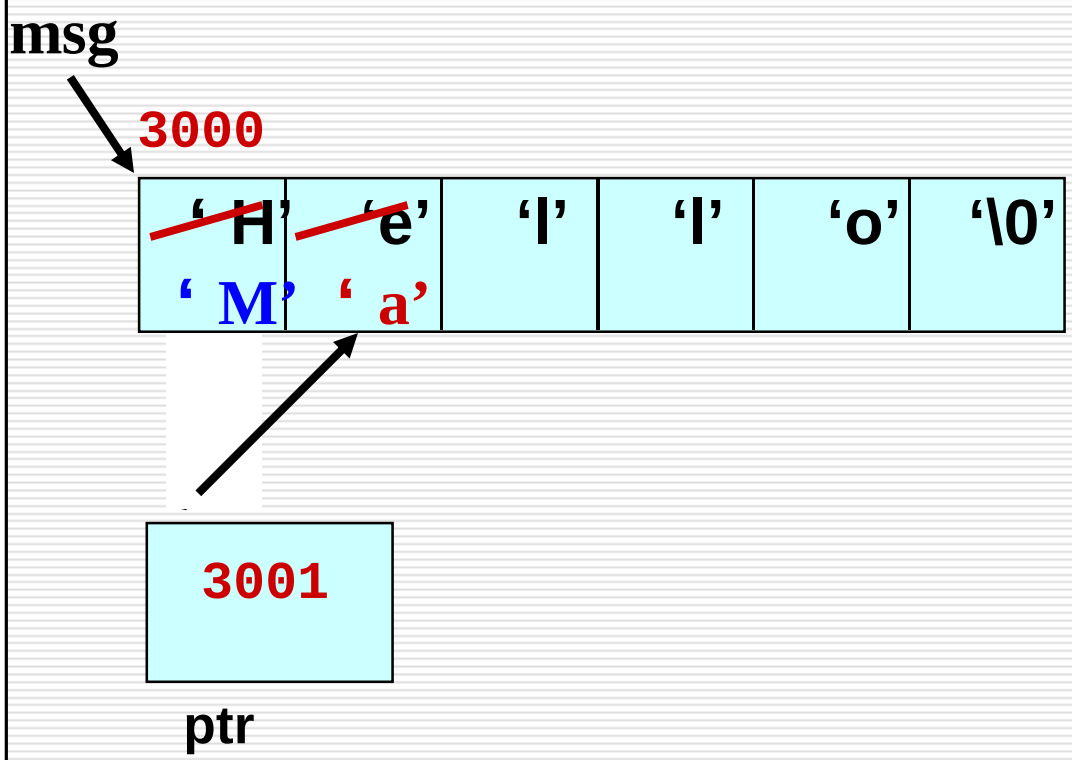
```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;

p = q;
```



// now both p and q point to *ch*

Pointers to Arrays

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

*ptr = ‘a’;

ptr = &msg[4];
// *ptr = o
```



Pointers and Constants

```
char s[] = “Hello”;
char* p = ‘Z’;
const char* pc = s;      // pointers to constant
pc[3] = ‘g’;             // error
pc = p;                  // ok

char *const cpc = s;     // constant pointer
cpc[3] = ‘a’;            // ok
cpc = p;                 // error
```

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

- 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 Variable 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;
}
```

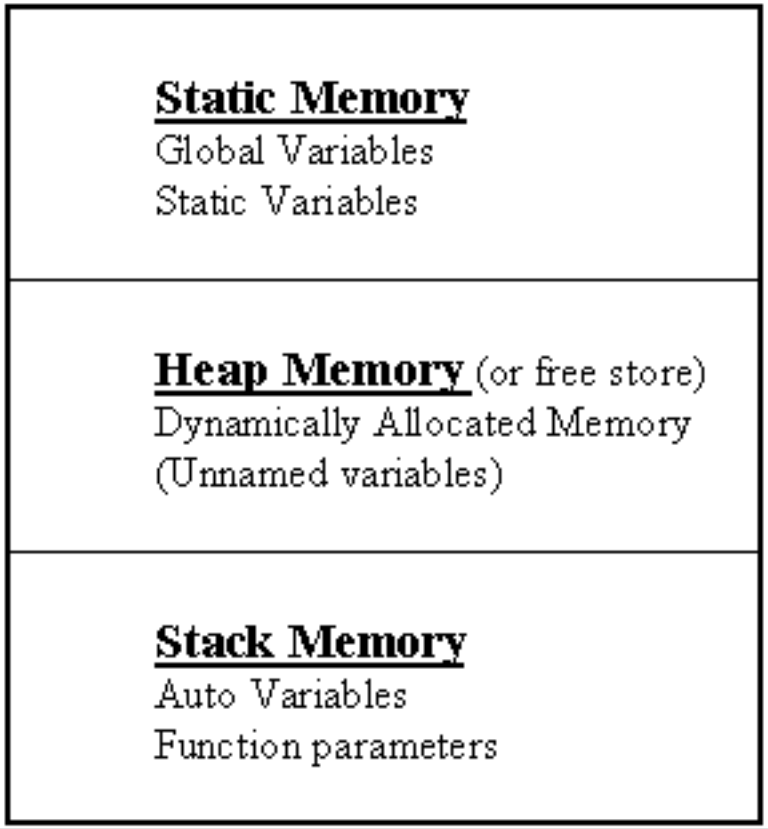
```
void p_swap(int *a, int *b)
{
  int temp;
  temp = *a;      (2)
  *a = *b;        (3)
  *b = temp;

}
```

```
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:

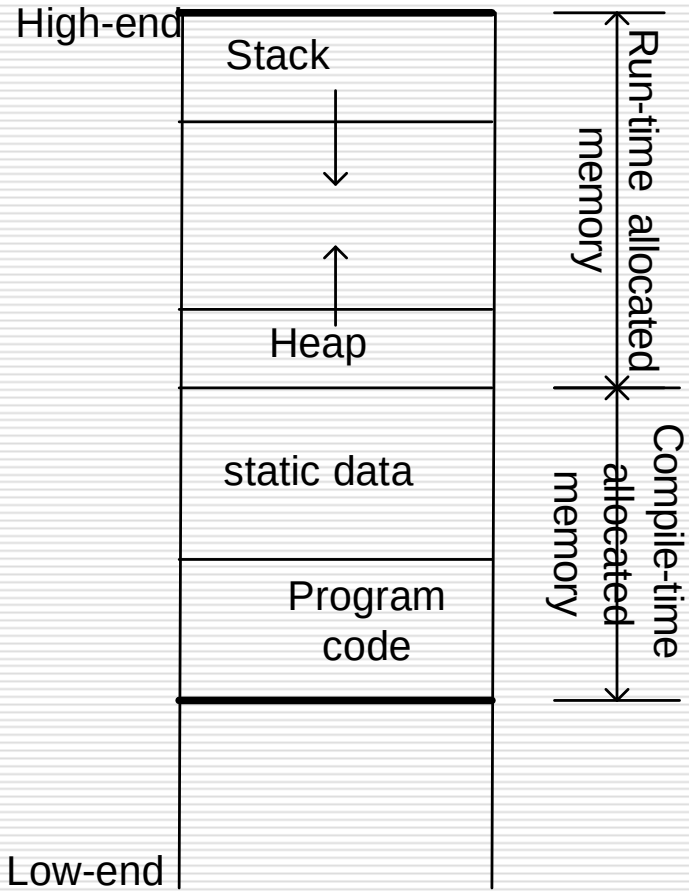
- **Static memory** - where global and static variables live
- **Heap memory -** dynamically allocated at execution time
  - "managed" memory accessed using pointers
- **Stack memory -** used by automatic variables



Three 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 []
```

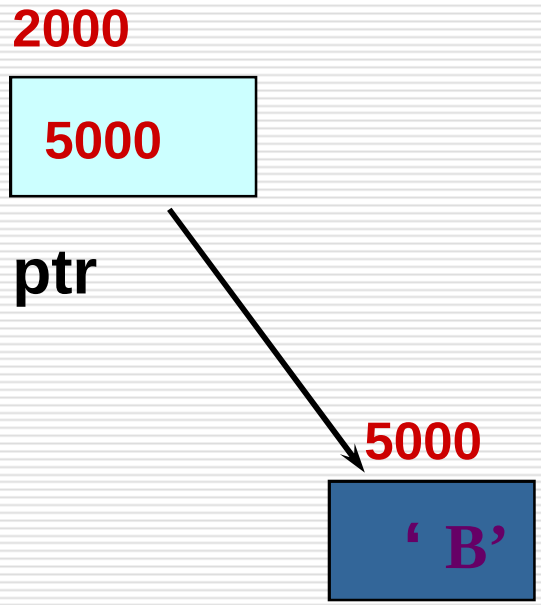
- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- Otherwise, program terminates with error message.
- The dynamically allocated object exists until the delete operator destroys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

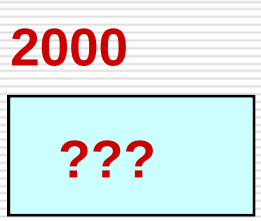
```
delete [] Pointer
```

- 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;
```

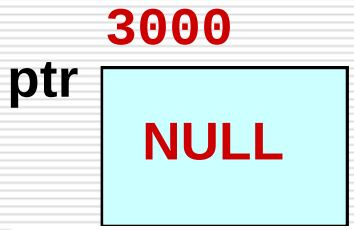


NOTE:

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
```

Take Home Message

- Be aware of where a pointer points to, and what is the size of that space.
- Have the same information in mind when you use reference variables.
- Always check if a pointer points to NULL before accessing it.

Hint for Lab #1

- How to parse the string from user input?
  - char *strtok (char *s, const char *delim);
  - strtok parses string s into tokens. The first call should have s as the first element
  - Subsequent calls should have the first argument set to NULL
- How to convert a character number to an integer?
  - int atoi (const char *nptr)
  - atoi converts the initial portion of the string pointed by nptr to int.