C++ Memory and Pointer

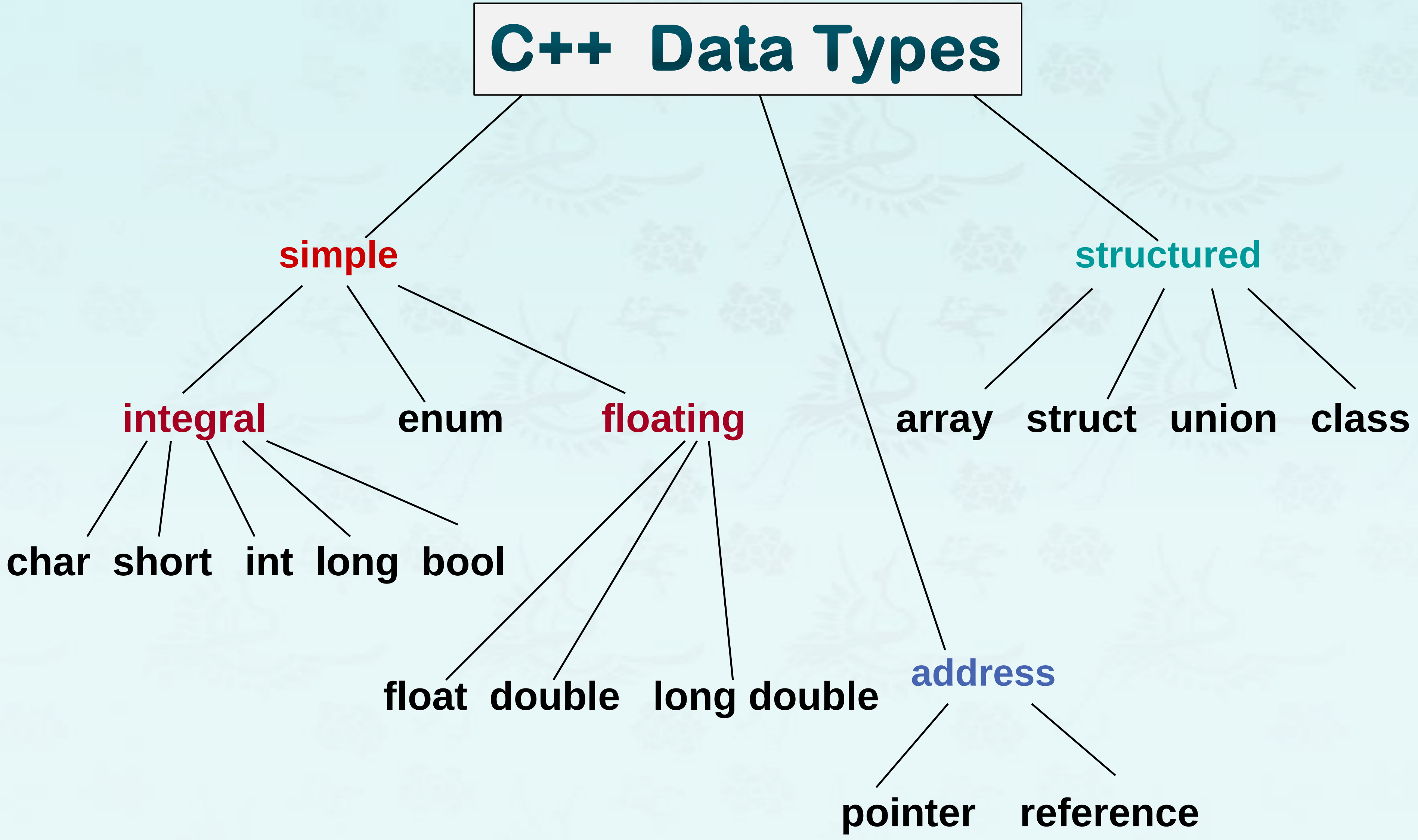
Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

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
- Quiz
- Lab or Homework

C++ Data Types



3

Addresses in Memory

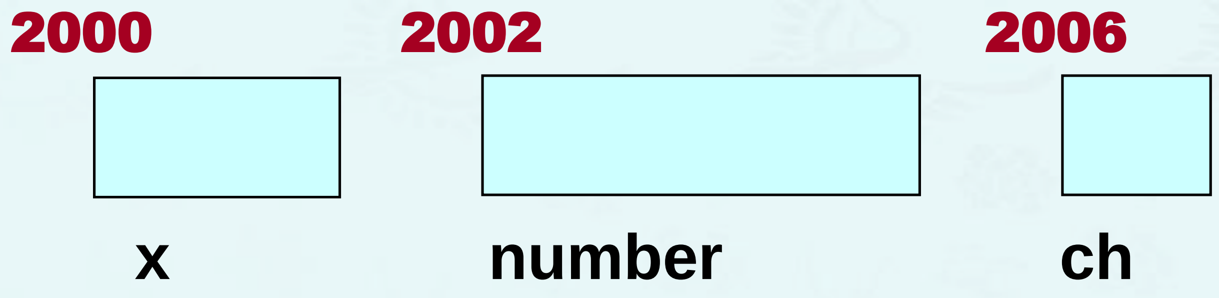
- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

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

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

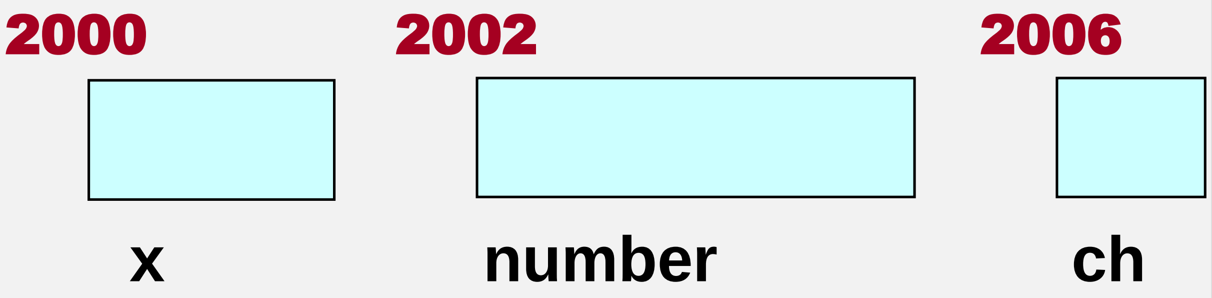
```
int       x;
float     number;
char      ch;
```



Obtaining Memory Addresses

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

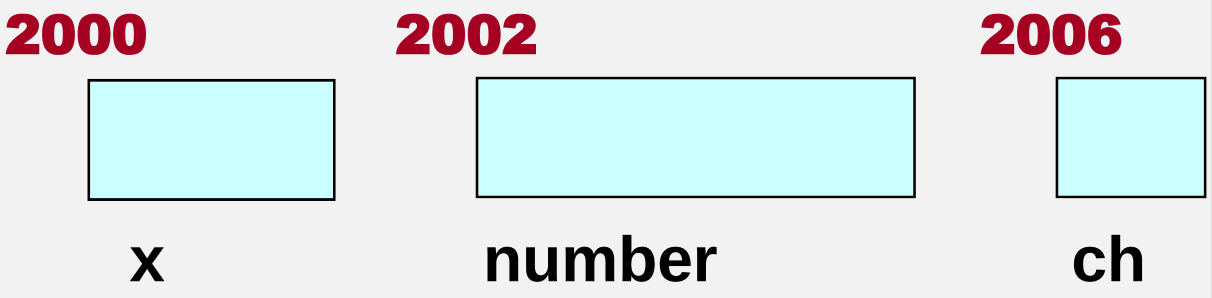
```
int     x;
float   number;
char    ch;
```



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?

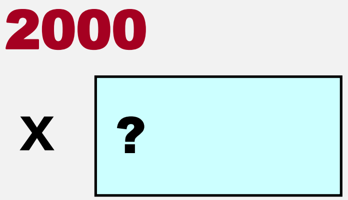
- A pointer variable is a variable whose value is **the address** of a location in memory.

- 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;
```



Using a Pointer Variable

```
int  x;
x = 12;
```

2000

x | 12

Using a Pointer Variable

```
int  x;
x = 12;

int  *ptr;
```

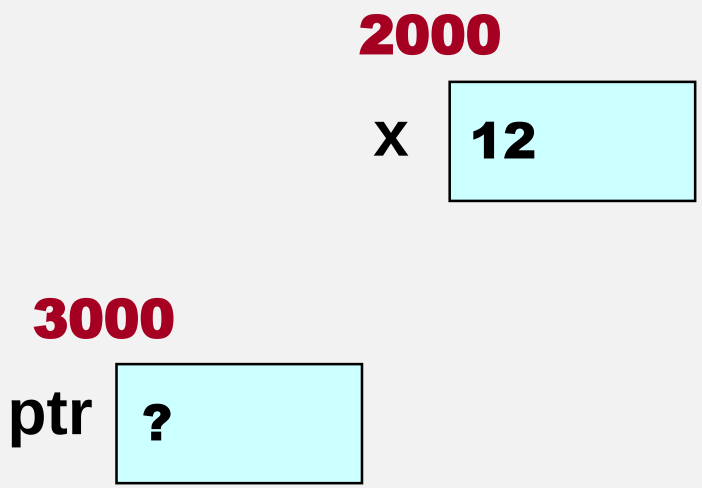
2000

x | 12

- NOTE: Because ptr holds the address of x, we say that ptr “points to” x

Using a Pointer Variable

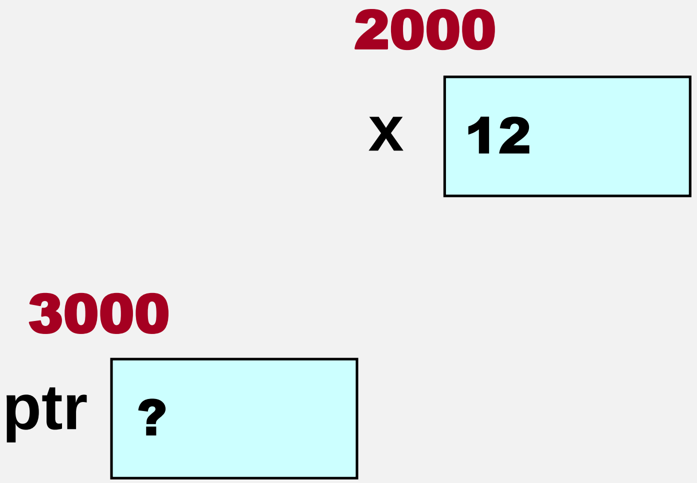
```
int  x;
x = 12;

int  *ptr;
```



- NOTE: Because ptr holds the address of x, we say that ptr “points to” x

Using a Pointer Variable

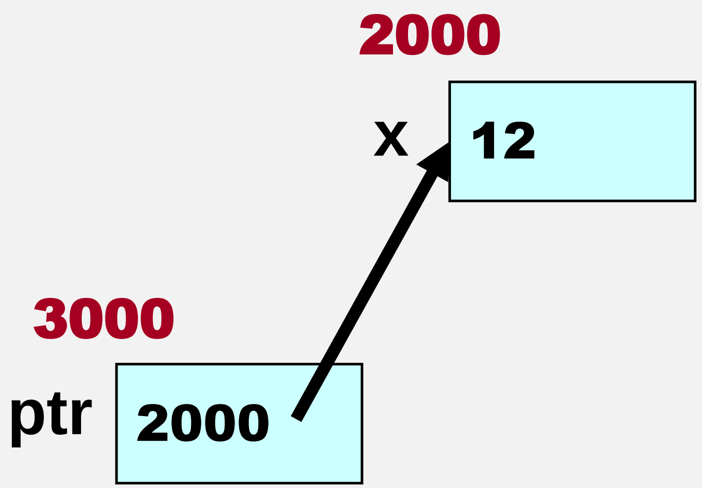
```
int  x;
x = 12;

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



- NOTE: Because ptr holds the address of x, we say that ptr “points to” x

Using a Pointer Variable

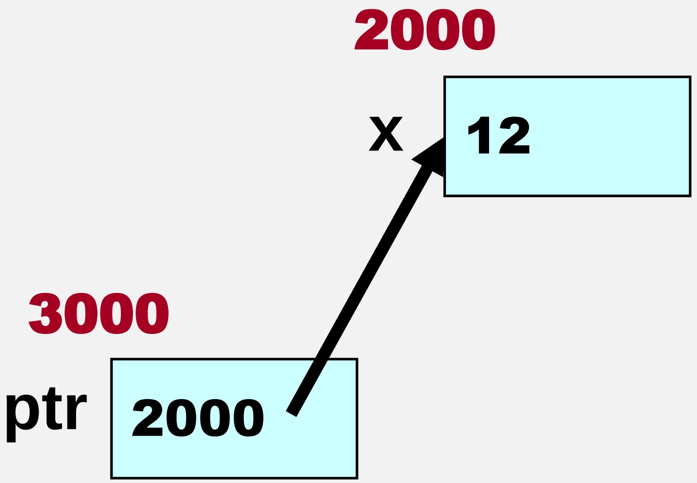
```
int  x;
x = 12;

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



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

Using the Dereference Operator *

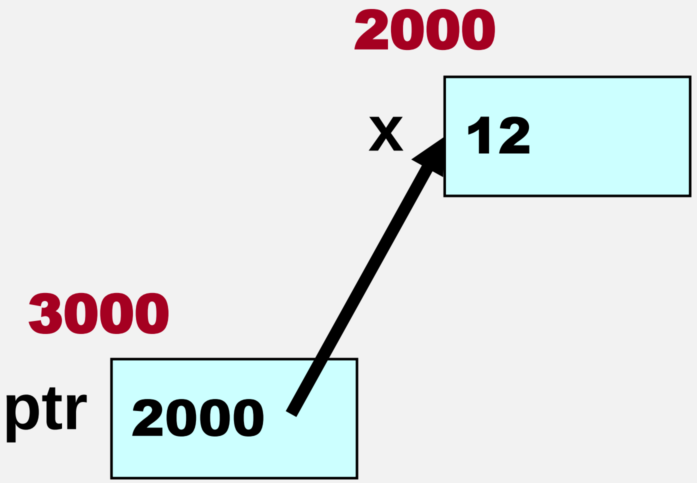
```
int  x;
x = 12;

int  *ptr;
ptr = &x;

cout << *ptr
```



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

Using the Dereference Operator *

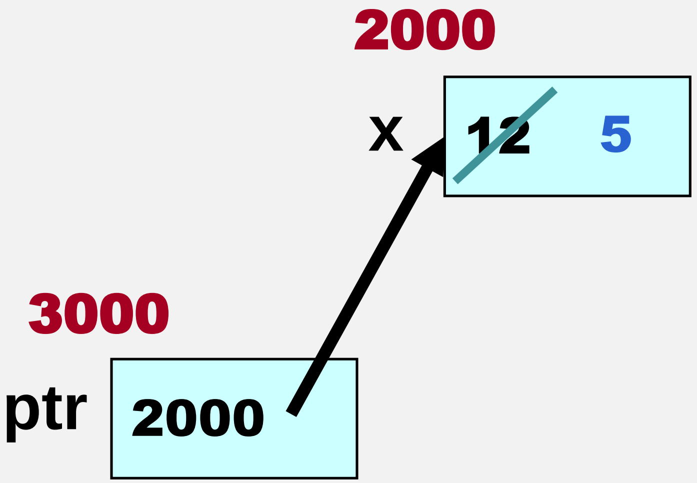
```
int  x;
x = 12;

int  *ptr;
ptr = &x;

*ptr = 5;
```



- NOTE: changes the value at the address ptr points to 5

Using the Dereference Operator *

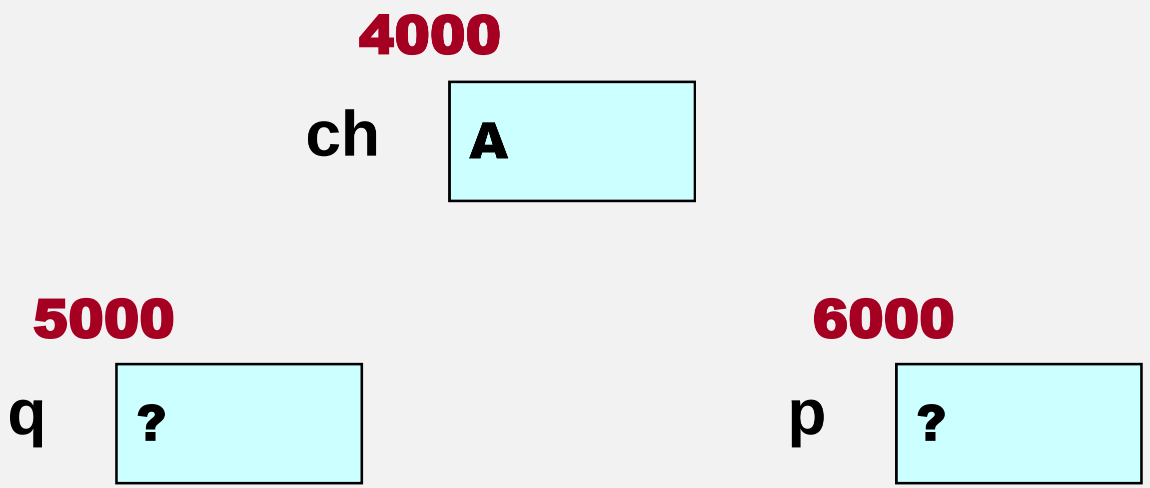
```
int  x;
x = 12;

int  *ptr;
ptr = &x;

*ptr = 5;
```



- NOTE: 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;
```



- NOTE: Complete the diagram and fix it if necessary.

Recall that . . .

```
char  str[8];
```

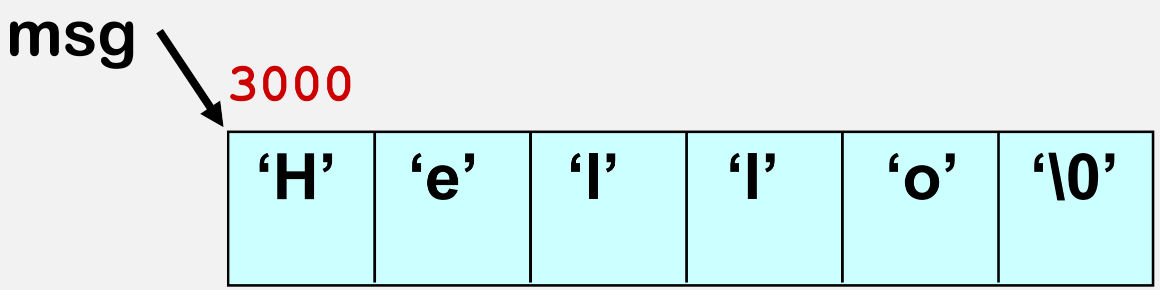
- `str` is the base address of the array.
- We say `str` is **a pointer** because its value is an address.
- It is a pointer constant because the value of `str` itself cannot be changed by assignment. It "points" to the memory location of a char.

6000

	'H'	'e'	'l'	'l'	'o'	'\0'		
`str`	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

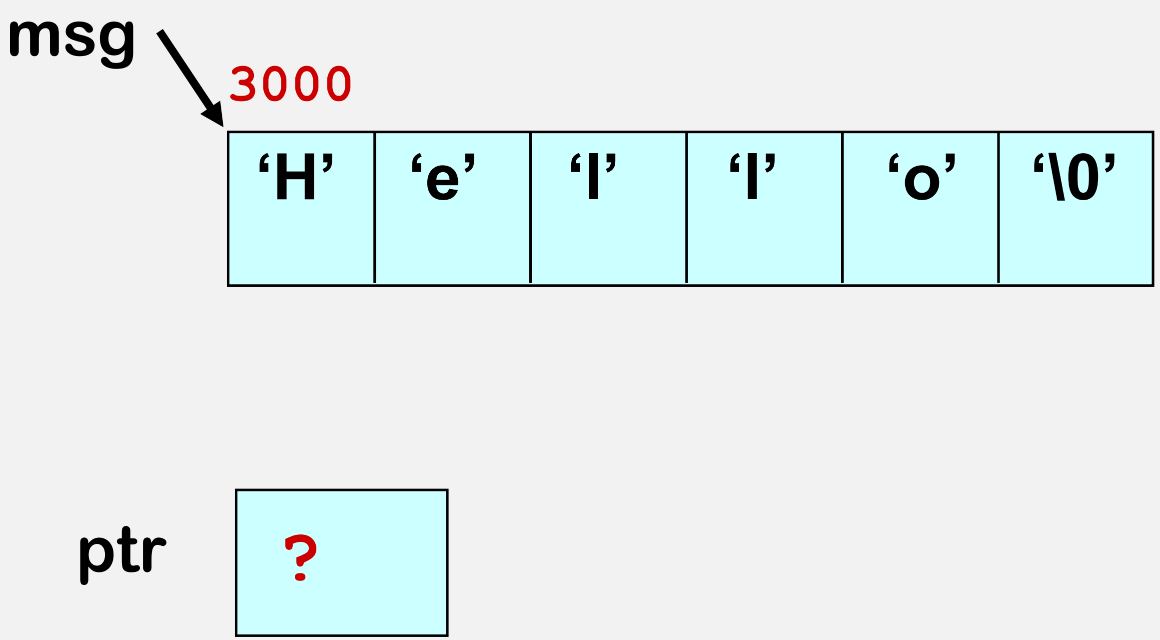
Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr    = msg;
*ptr   = 'M' ;
ptr++;

*ptr = 'a';
```



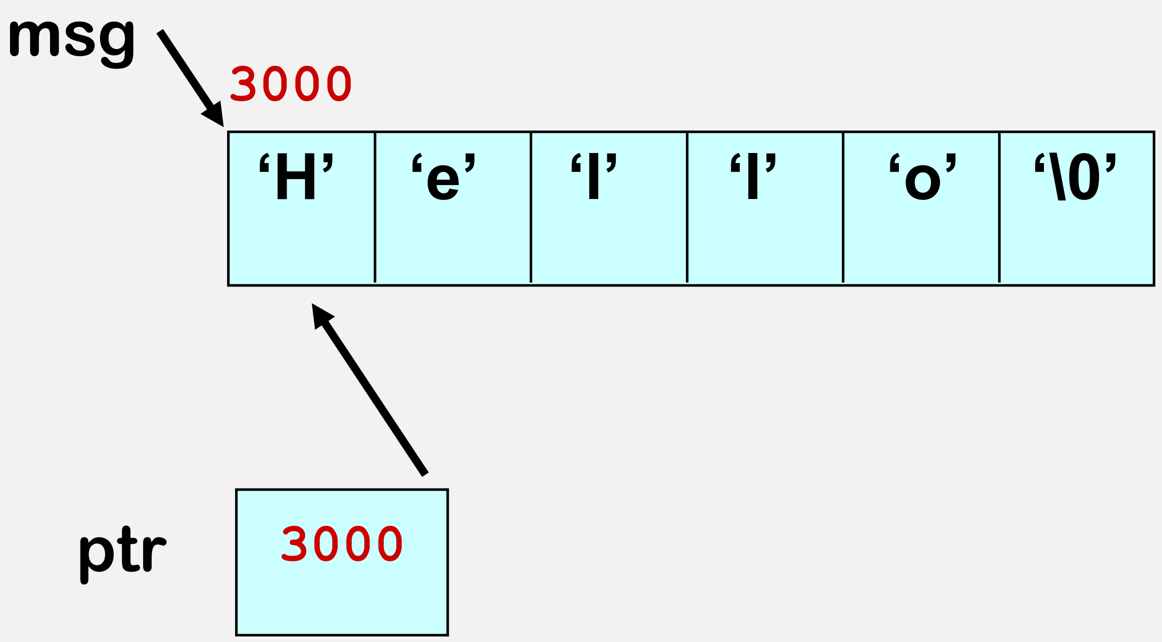
Using a Pointer to Access the Elements of a String

```
char    msg[] ="Hello";
char*   ptr;
ptr    = msg;
*ptr   = 'M' ;
ptr++;

*ptr = 'a';
```



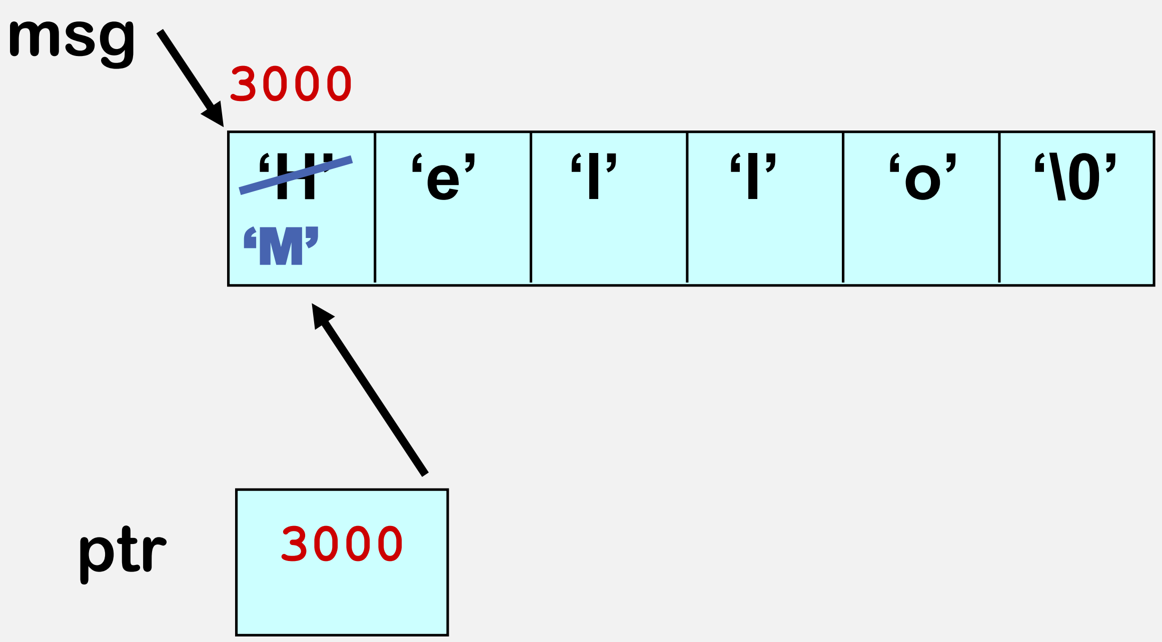
Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr   = msg;
*ptr  = 'M' ;
ptr++;

*ptr = 'a';
```



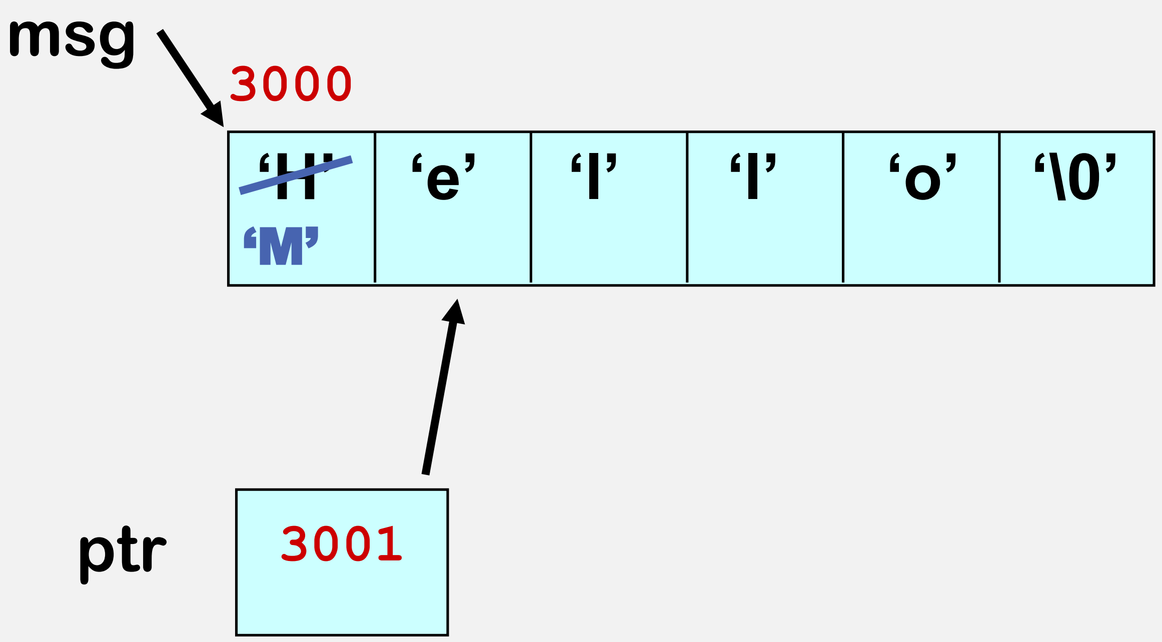
Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr    = msg;
*ptr   = 'M' ;
ptr++;

*ptr = 'a';
```



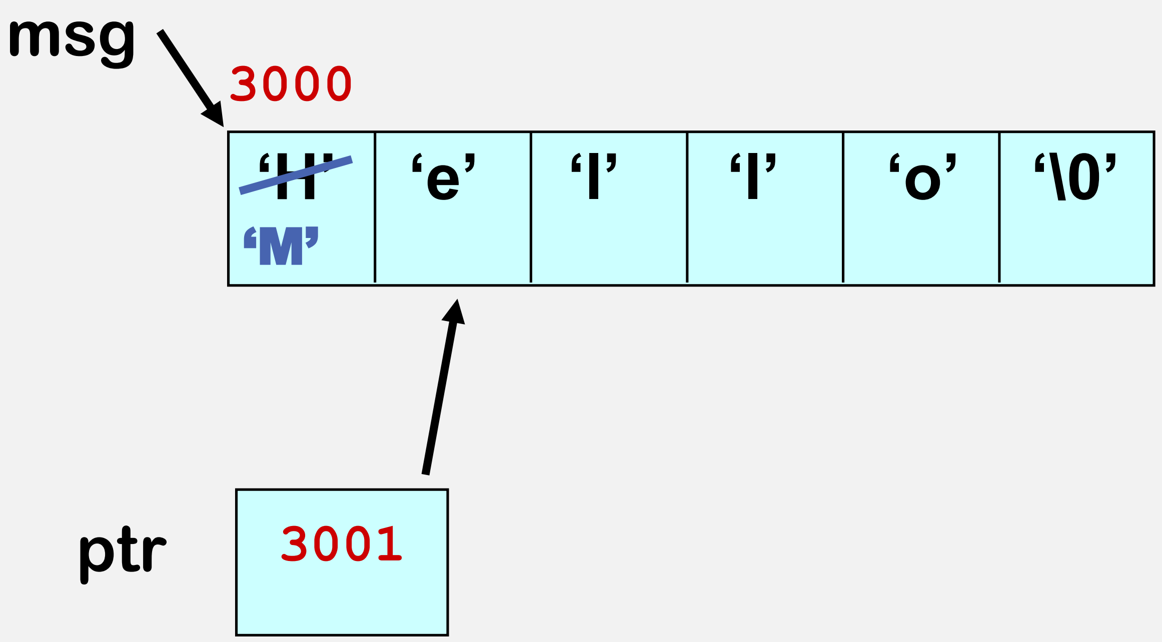
Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr    = msg;
*ptr   = 'M' ;
ptr++;

*ptr = 'a';
```



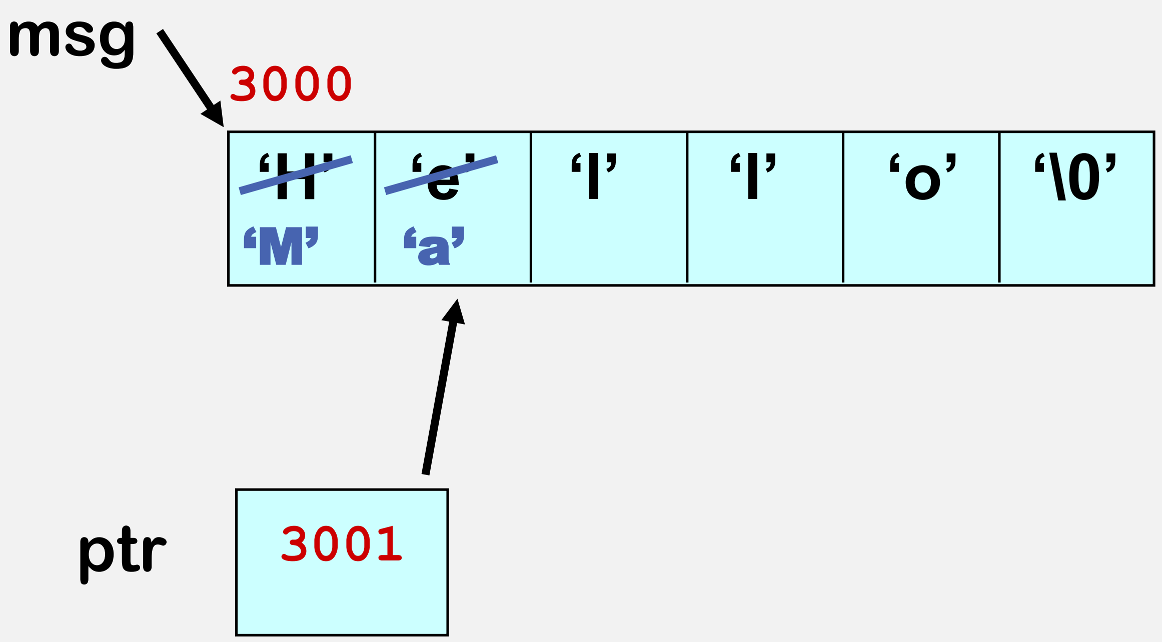
Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr    = msg;
*ptr   = 'M' ;
ptr++;

*ptr = 'a';
```



Using a Pointer to Access the Elements of a String

```
char   msg[] ="Hello";
char*  ptr;
ptr   = msg;
*ptr  = 'M' ;
ptr++;

*ptr = 'a';
```



Reference Variables in C++, but not in C

- 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
```

Reference Variables in C++, but not in C

- 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
```

Reference Variables in C++, but not in C

- 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

No overloading in C

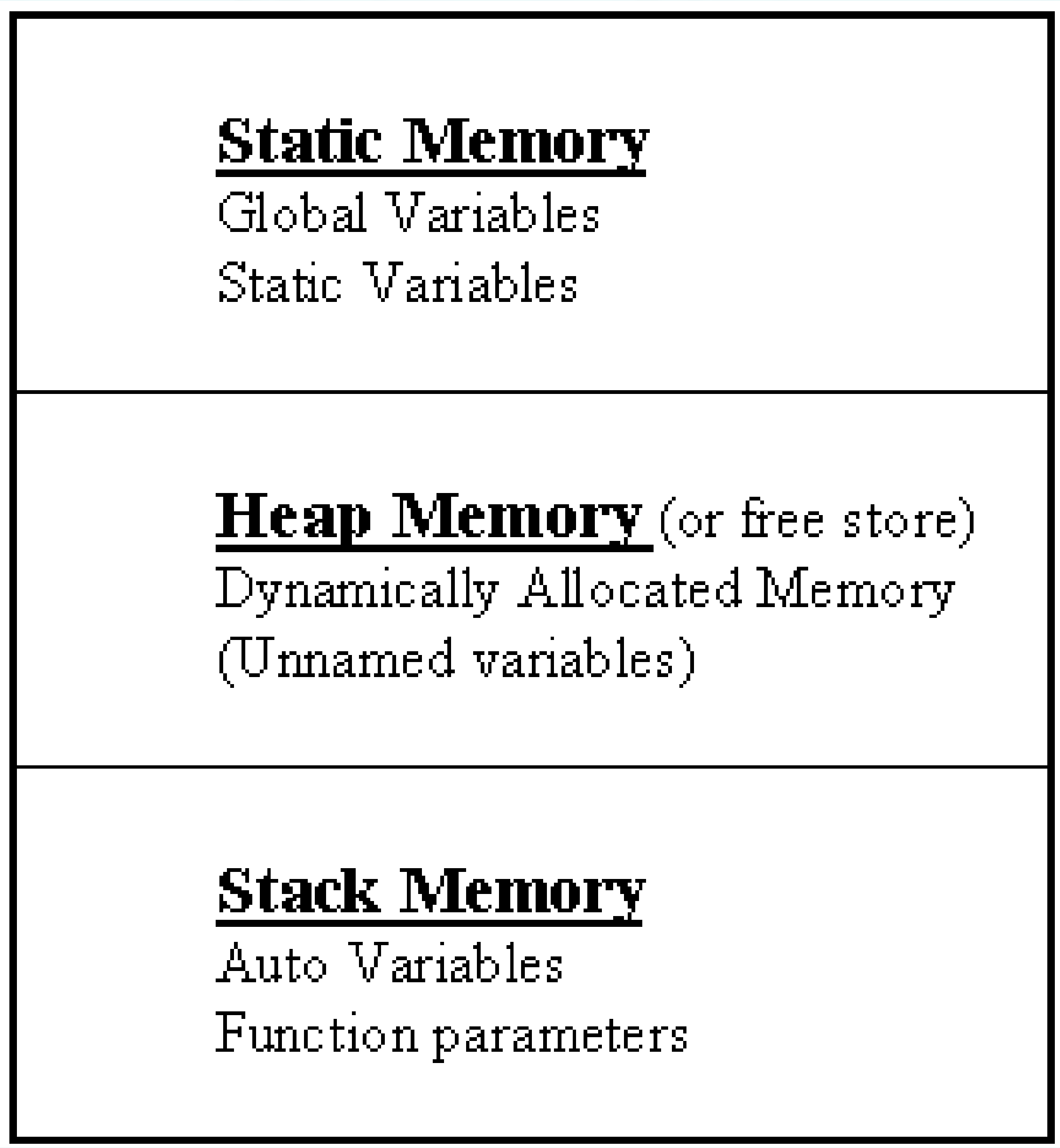
```
#include <iostream.h>

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

int main() {
  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;
}
```

Why C++ is better

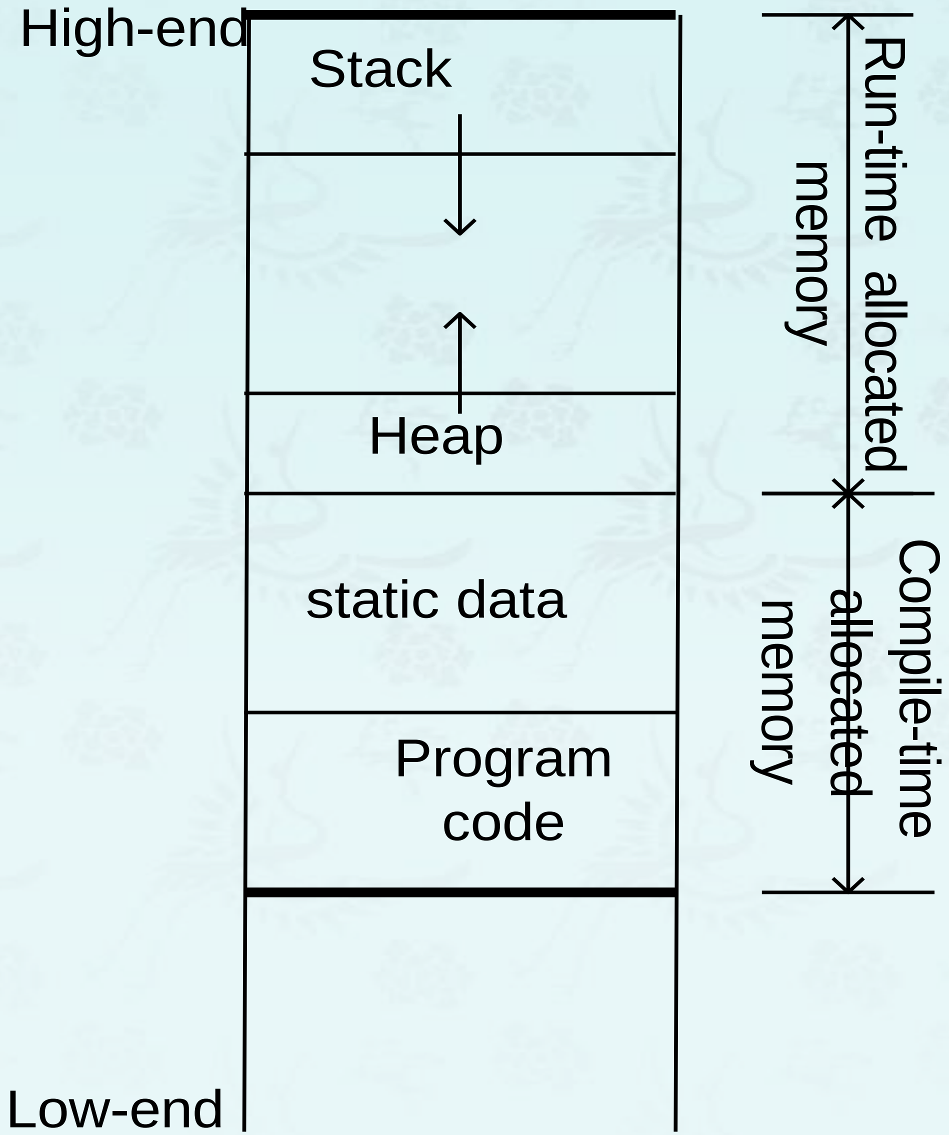
- In C and C++, three types of memory are used by programs:
- **Static memory**
  - where global and static variables live
- **Heap memory**
  - dynamically allocated at execution time
  - "managed" memory accessed using pointers
- **Stack memory**
  - used by automatic variables



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

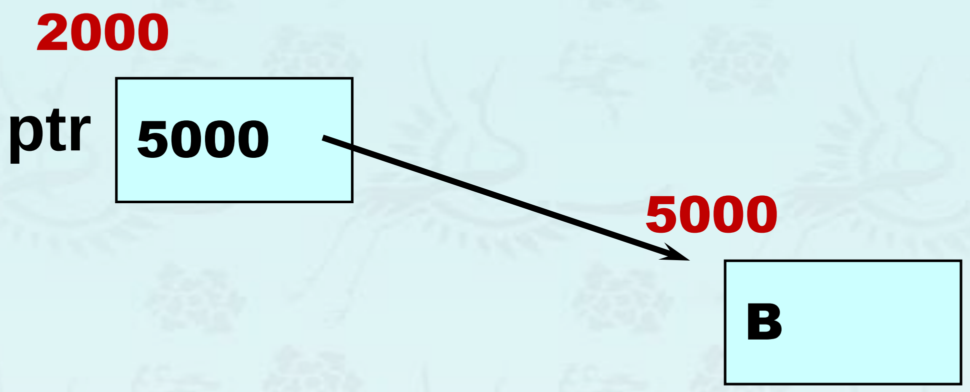
```
new DataType
```

```
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**

Examples Using new & delete

```
int *pi = new int;                    // pi points to uninitialized int
int *pi = new int(77);                // which pi points has value 77
string *ps = new string;              // empty string

int *pia = new int[10];               // block of ten uninitialized ints
int *pia = new int[10]();             // block of ten ints values initialized to 0

string *psa = new string[5];                 // block of 5 empty strings
string *psa = new string[5]();               // block of 5 empty strings
int    *pia = new int[5]{0, 1, 2, 3, 4};     // block of 5 ints initialized
string *psa = new string[2]{"a", "the"};     // block of 2 strings initialized

delete   pi;
delete[] pia;
```

new vs. malloc()

new	malloc()
new is an operator.	malloc is a library function.
It calls the constructor.	It does not call the constructor.
It returns exact data type if memory is available.	It returns the void * if memory is available.
It throws bad_alloc exception on failure. Use `nothrow` for `nullptr`.	It returns `nullptr` on failure.
It can be overridden.	It cannot be overridden.
In which memory allocated from the heap.	In which memory allocated from the heap.
Size is calculated by the compiler.	Need to pass the size.

NOTE: We learn how to use both malloc() as well as new first. Once we get familiar with them, then we rather start using new and delete operators more and more later in this course.

Dynamic Memory Allocation

- *In C*, functions such as malloc() and free() are used to dynamically allocate and deallocate 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**

The NULL/nullptr Pointer

- There is a pointer constant called the "null pointer" denoted by NULL/nullptr.
- NULL is int type 0 in C/C++, but nullptr is std::nullptr_t type.
- NOTE: It is an error to dereference a pointer whose value is NULL or nullptr. Such an error may cause your program to crash, or behave erratically. It is the programmer's job to check for this.

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

Operator delete Syntax

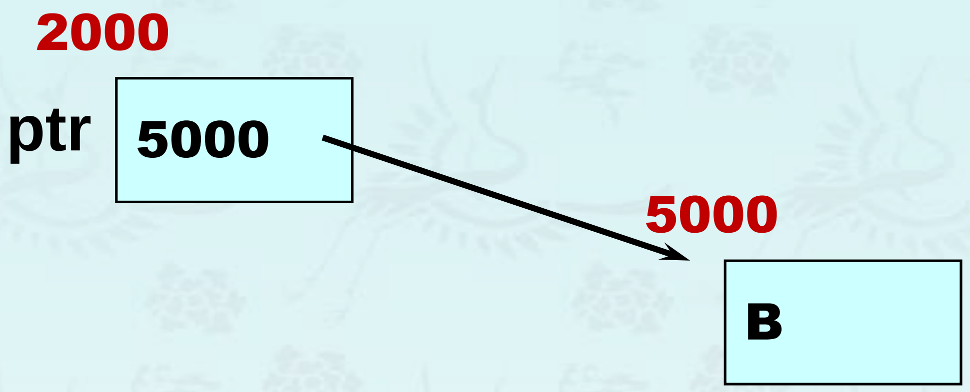
```
delete PointerVariable
```

```
delete [ ] PointerVariable
```

- 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 nullptr.
- Square brackets are used with delete to deallocate a dynamically allocated array.

Operator **delete**

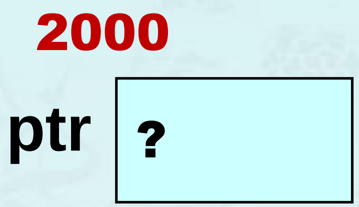
```
char*  ptr;

ptr = new char;

*ptr = 'B';

delete ptr;
```



NOTE: **delete** deallocates the memory pointed to by ptr

Operator **delete**

```
char*  ptr;

ptr = new char;

*ptr = 'B';

delete ptr;
```



NOTE: **delete** deallocates the memory pointed to by ptr

Example: Operator delete

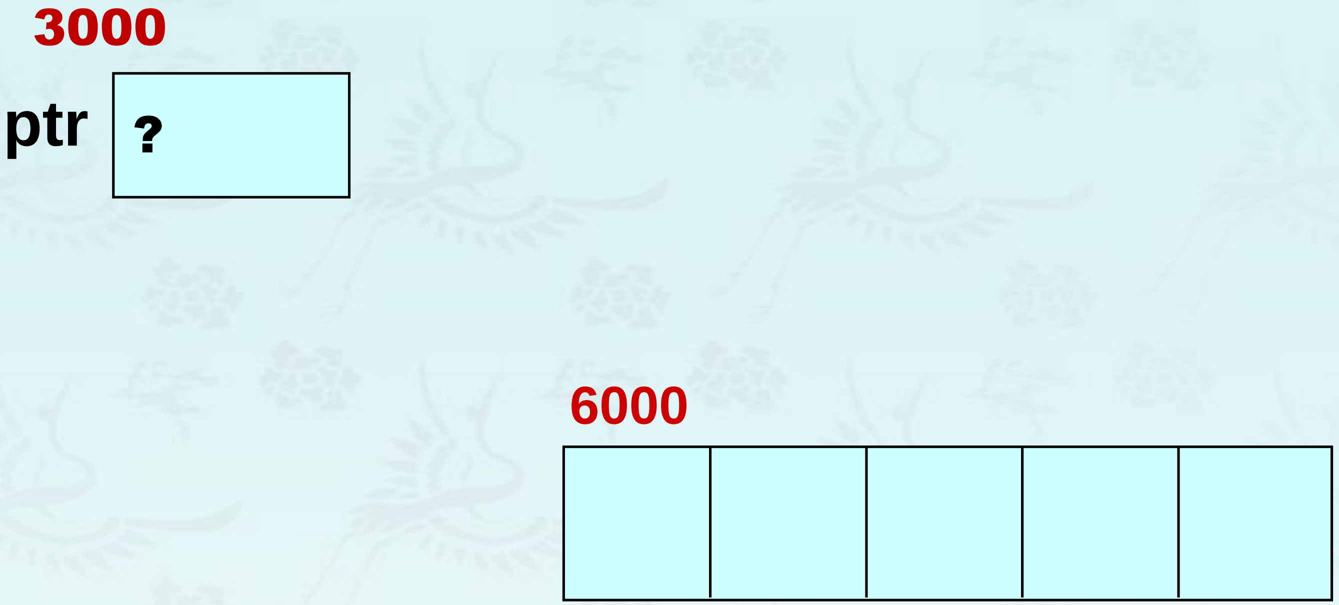
```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```

3000

ptr | ? |

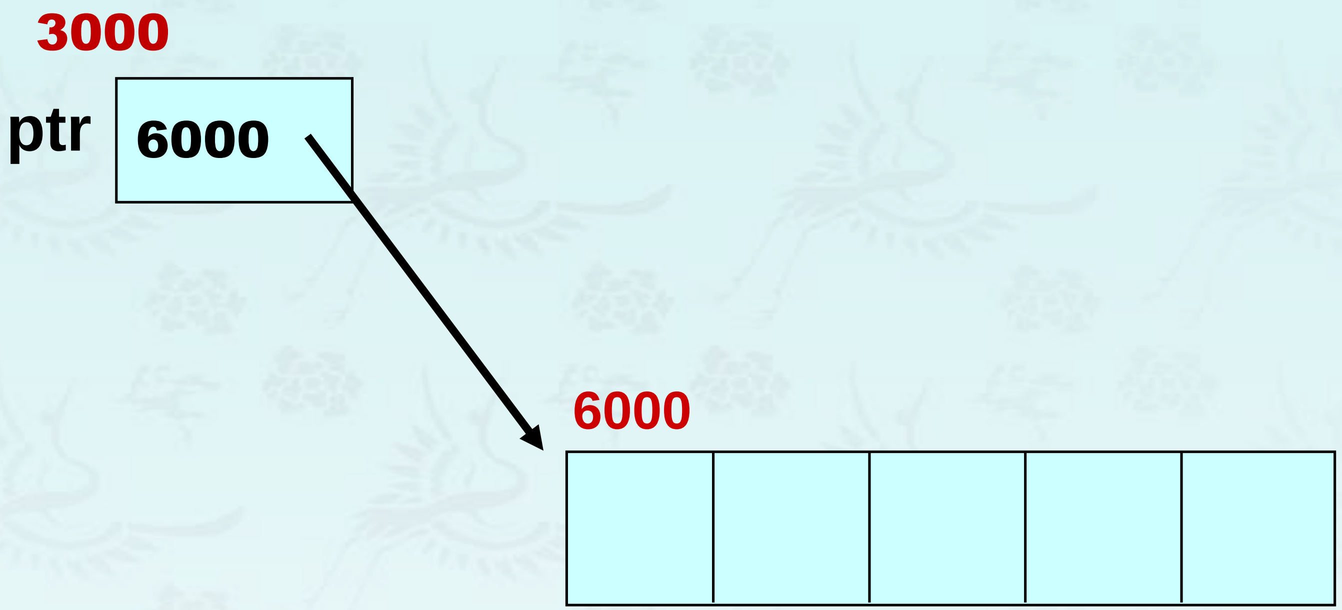
Example: Operator **delete**

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```



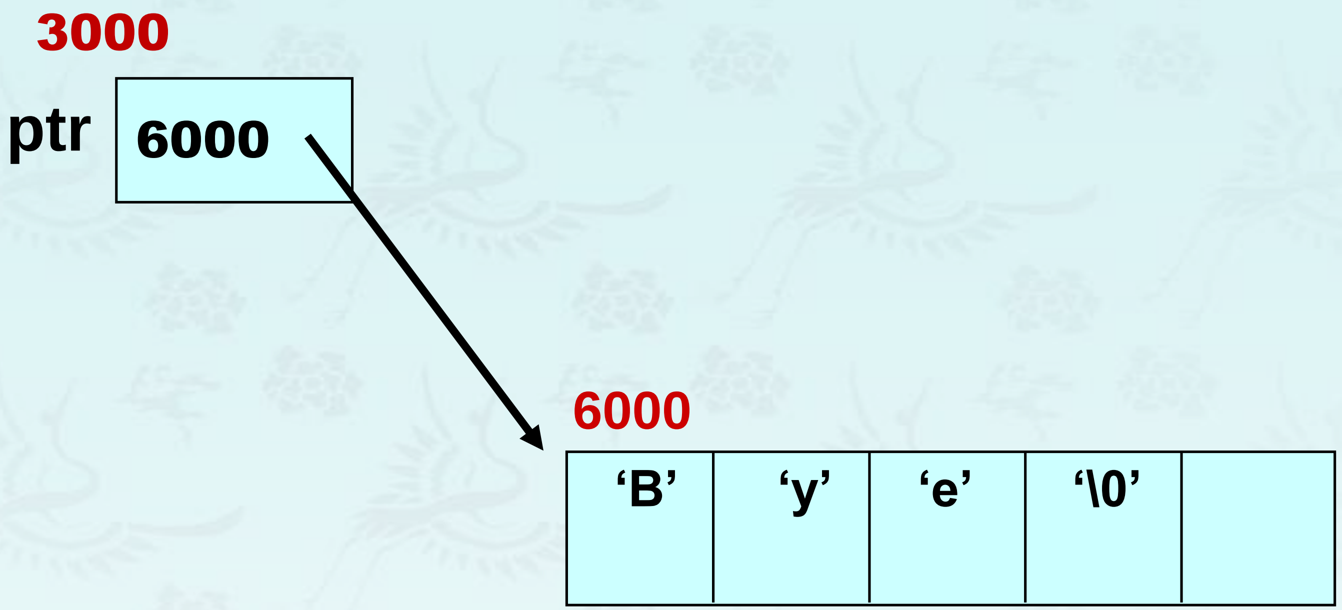
Example: Operator delete

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```



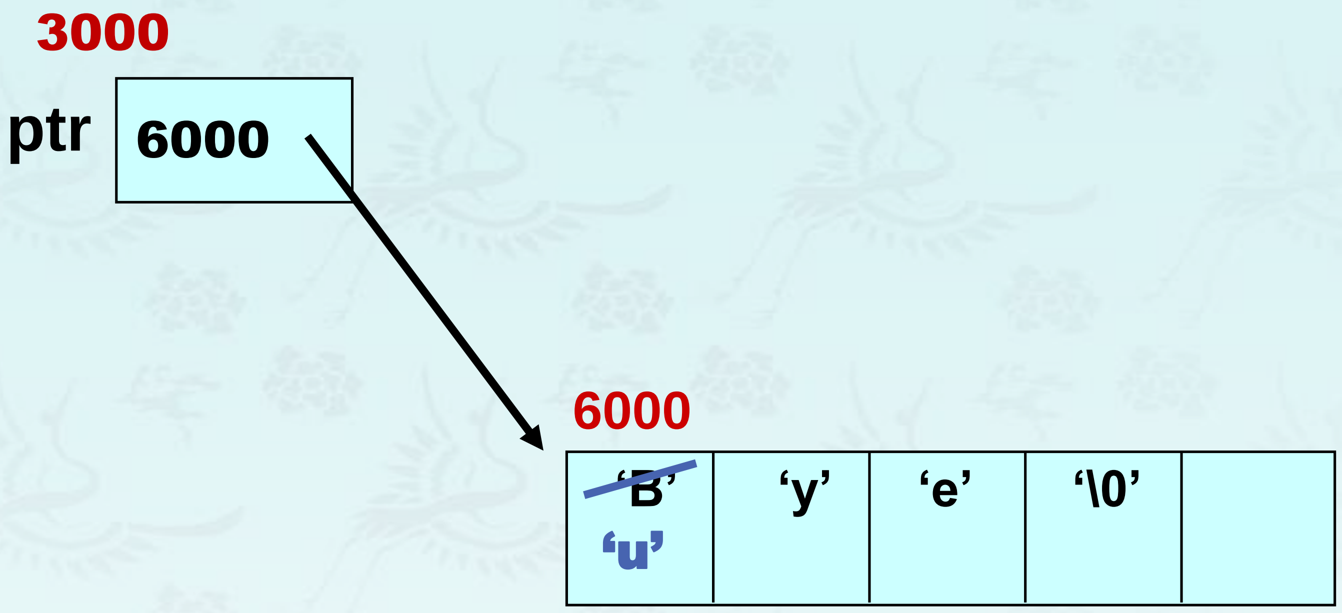
Example: Operator delete

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```



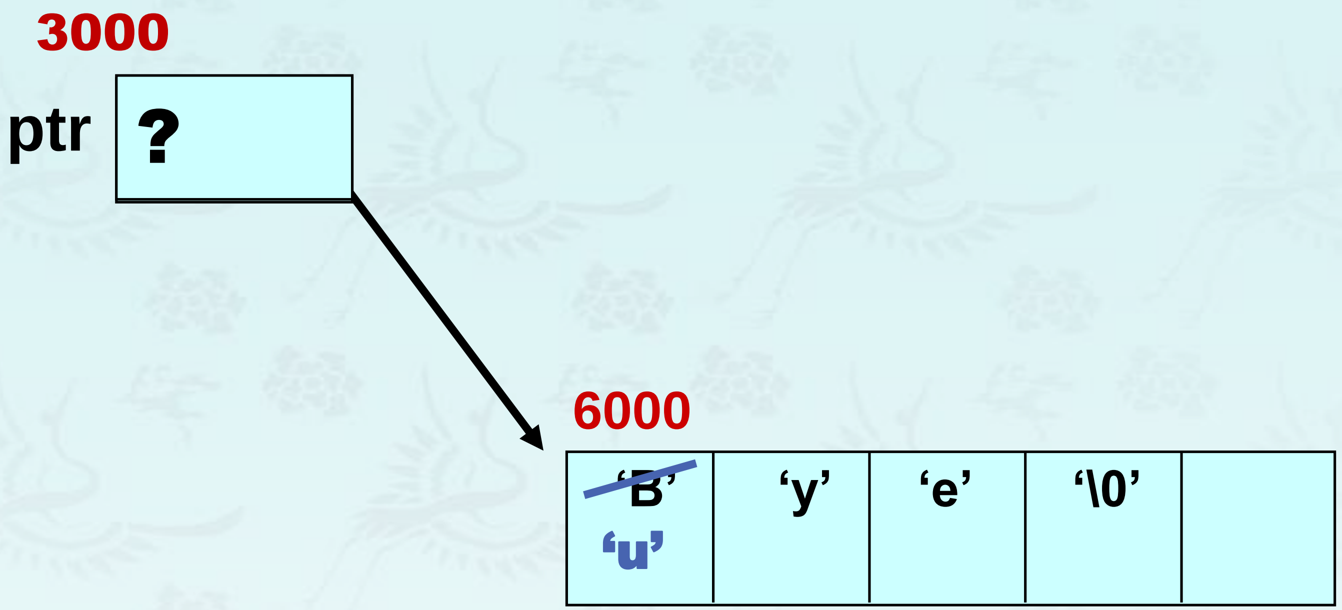
Example: Operator **delete**

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```



Example: Operator **delete**

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```



NOTE:

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

Example: Operator delete

```
char  *ptr;

ptr = new char[5];

strcpy(ptr, "Bye");

ptr[0] = 'u';

delete [] ptr;

ptr = nullptr;
```

3000

ptr NULL

NOTE:

- 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 nullptr before accessing it. For example,

```
char *ptr = new (nothrow) char[5];
assert(ptr != nullptr);
```