Variables and Memory

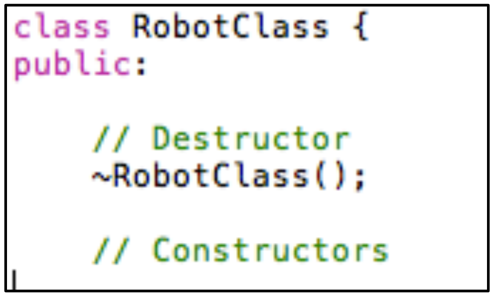
CSCI3081

Program Design and Development

Destructors

- Every Class has a destructor (and only 1).
- If you don't define one, the compiler will.
- The destructor is called
  - by you
  - OR any time a class object is destroyed (goes out of scope).

- What should it do ?
  - Free memory (delete any new objects).
  - Close handles to resources (like files).
  - Any other clean-up required.



```
class RobotClass {
public:

    // Destructor
    ~RobotClass();

    // Constructors
```

```
// Define the Destructor
RobotClass::~RobotClass() {
     printf("Destroying Object\n");
}

// Instantiate the Object
RobotClass robot;

// Explicitely Call the Destructor
robot.~RobotClass();
```

Variables, Pointers, and Memory

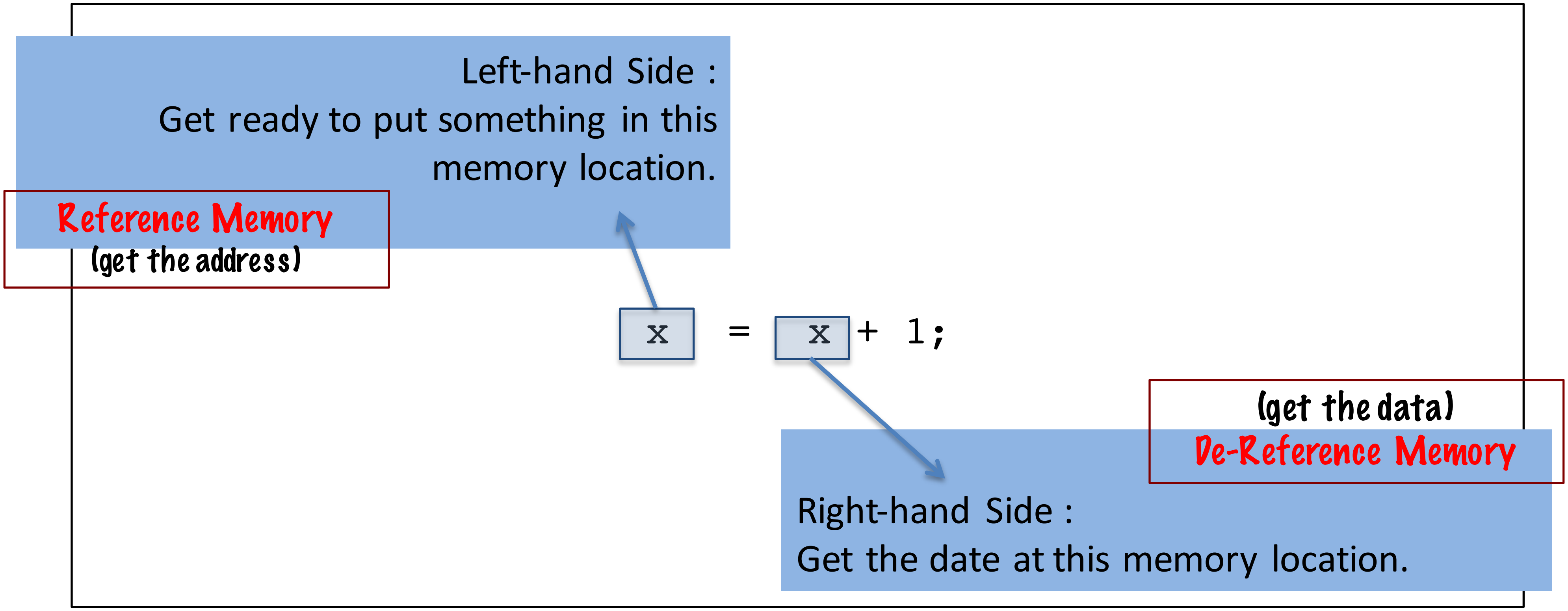
- Variables and Memory
- Parameter Passing and Return Values
- Pointers and References
  - Defining, Declaring, Initializing
  - Reference and Dereference Operators
  - Parameter Passing
  - NULL and void
- Hidden Pointers – arrays and strings
- Odds and Ends about Pointers

Variables and Memory

```
int x;
x = 25;
x = x + 1;
```

fill memory location “x” with 25

take data in memory location “x”, add 1,
fill memory location “x” with results



Left-hand Side :
Get ready to put something in this memory location.

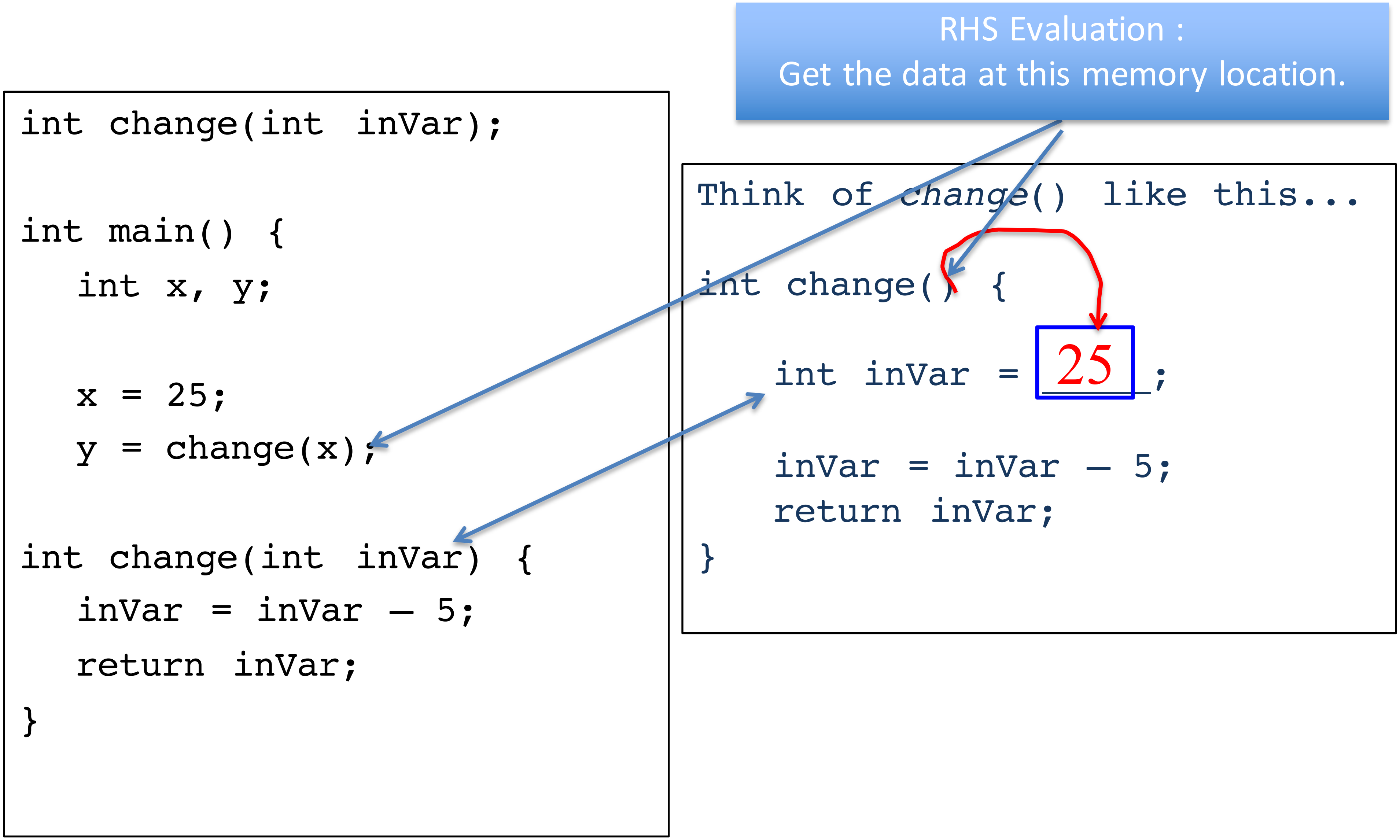
Reference Memory
(get the address)

x = x + 1;

(get the data)
De-Reference Memory

Right-hand Side :
Get the date at this memory location.

Parameter Passing



RHS Evaluation :
Get the data at this memory location.

```
int change(int  inVar);

int main()  {
   int x,  y;

   x  =  25;
   y  =  change(x);

int change(int  inVar)  {
   inVar  =  inVar  –  5;
   return  inVar;
}
```

```
Think  of  change()  like  this...

int  change()  {

    int  inVar  =  25;

    inVar  =  inVar  –  5;
    return  inVar;
}
```

Pointers and References :
Declaring, Defining, and Initializing

Tell the compiler

"I want to use addresses too!"

(int *) is the type.

(double *) is the type.

```
int *px;                // decl & def of pointer-integer
int* py;                // decl & def of pointer-integer
double *pz = &z;        // decl, def & init of pointer-double
```

"&" makes this an address (like LHS).

```
px = &x;                // initialize px to the address of x
*px = 50;               // set x equal to 50;
```

```
~~int &rw;~~               // ILLEGAL. Must be initialized.
int &rx = x;            // decl, def, & init of reference-int
```

(int &) is the type.

```
rx = 25;                // set x equal to 25;
```

Parameter Passing with Pointers and References

```
void swap(int A, int B);

int main() {
   int x, y;

   x = 25;
   y = 10;
   swap(x,y);

void swap(int A, int B) {
   int temp = A;
   A = B;
   B = temp;
}
```

RHS Evaluation :
Get the data at this memory location.

```
Think of swap() like this...

void swap(int A, int B) {

    int A = 25;

    int B = 10;

    int temp = A;
    A = B;
    B = temp;
}
```

swap() with pointers and referenes

swap() with pointers

(pass-by-value)

```
void swap(int* A, int* B);

int main() {
    int ____ x = ____ ;
    int ____ y = ____ ;
    swap(____x, ____y);
}

void swap(int* A, int* B) {
    int __ temp;
    ___temp = ___A;
    ___A = ____B;
    ___B = ____temp;
}
```

x=25;
y=10;

the only way in C
(references do not
exist)

swap() with references

(pass-by-reference)

```
void swap(int& A, int& B);

int main() {
    int ____ x =  ____ ;
    int ____ y =  ____ ;
    swap(____x, ____y);
}

void swap(int& A, int& B) {
    int __ temp;
    ___temp = ____A;
    ___A = ____B;
    ___B = ____temp;
}
```

swap() with pointers and referenes

swap() with pointers

(pass-by-value)

```
void swap(int* A, int* B);


int main() {
    int ____ x = 25;
    int ____ y = 10;
    swap( &x, &y);
}
```

the only way in C (references do not exist)

&x = 0x02

&y = 0x06

```
void swap(int* A, int* B) {
    int temp;
    temp = *A;
    *A = *B;
    *B = temp;
}
```

A = 0x02

B = 0x06

swap() with references

(pass-by-reference)

```
void swap(int& A, int& B);


int main() {
    int ____ x = 25 ;
    int ____ y = 10 ;
    swap(____x, ____y);
}


void swap(int& A, int& B) {
    int __ temp;
    ___temp = ____A;
    ___A = ____B;
    ___B = ____temp;
}
```

NULL

DANGEROUSchange()

```
void DANGEROUSchange(
        int* A, int* B );

int main() {
   int* px;
   int* py;
   DANGEROUSchange(px, py);
}

void DANGEROUSchange(
        int *A, int *B) {

   *px = 50;
   *py = 100;
}
```

fixing DANGEROUSchange()

```
if ((NULL == px) || (NULL == py))
    return;
```

```
if ((0 == px) || (0 == py))
   return;
```

```
if ( (!px) || (!py))
   return;
```

```
if ( px && py )
   // proceed with change
```

```
if ( px && (*px = 50) );
if ( py && (*py = 100) );
```

Variables, Pointers, and Memory

- Variables and Memory
- Parameter Passing and Return Values
- Pointers and References
  - Defining, Declaring, Initializing
  - Reference and Dereference Operators
  - Parameter Passing
  - NULL and void
- Hidden Pointers – arrays and strings
- Odds and Ends about Pointers

void pointers

variable types of objects

```
enum MyObjectType { circle, square };

struct {
     int length;
     bool filled;
     int area;
} SquareStruct;

struct {;
     int radius;
     bool filled;
     double area;
} CircleStruct;

int main() {
     SquareStruct mySquare;
     mySquare.length = 5;
     mySquare.filled = true;

     CircleStruct* myCircle = new CircleStruct;
     myCircle->radius = 3;
     myCircle->filled = false;

     setArea(&mySquare,square);
     setArea(myCircle,circle);
}
```

Two types of structures: circles and squares.

decl, def, init squareStruct

decl, def, init CircleStruct

setArea ("overloaded" param types)

setArea(void *object, MyObjectType type)

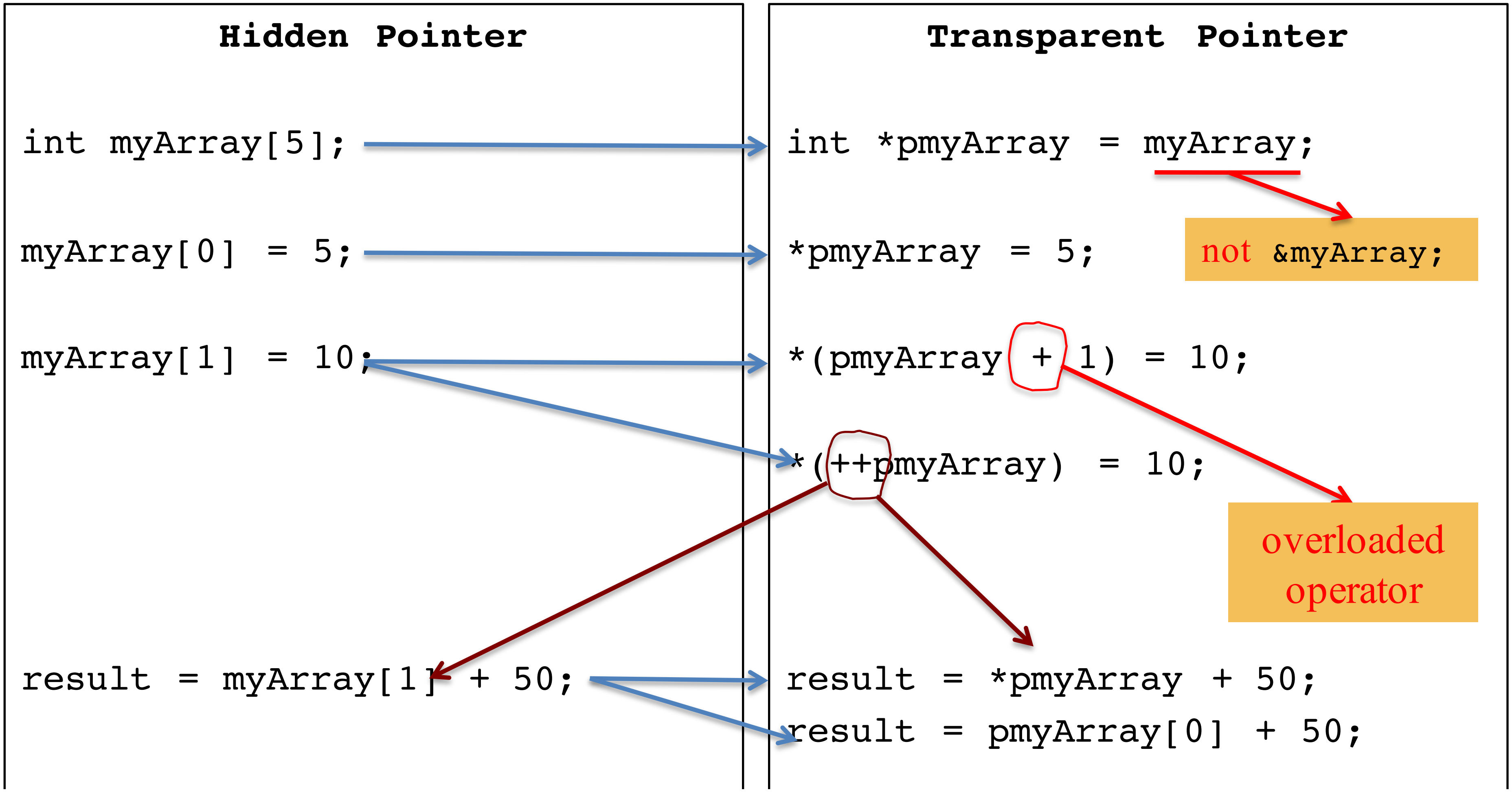
```
void setArea(void *myObject, MyObjectType type)
{
  SquareStruct* squareObject;
  CircleStruct* circleObject;

  switch (type) {

    case square:
      squareObject = (SquareStruct *)myObject;
      squareObject->area =
                 squareObject->length
                    * squareObject->length;
      break;

  case circle:
    circleObject = (CircleStruct *) myObject;
    circleObject->area =
        circleObject->radius * 3.14 * 2.0;
    break;
  }
  return;
```

You have to cast void pointers.

Arrays and Pointers

Internally, *arrays* are blocks of nondescript memory.

Arrays are de/referenced with pointers (but you don't see it).



Hidden Pointer

```
int myArray[5];

myArray[0] = 5;

myArray[1] = 10;

result = myArray[1] + 50;
```

Transparent Pointer

```
int *pmyArray = myArray;

*pmyArray = 5;

*(pmyArray + 1) = 10;

*(++pmyArray) = 10;

result = *pmyArray + 50;
result = pmyArray[0] + 50;
```

not &myArray;

overloaded operator

Arrays and Pointers and Dynamic Allocation

DYNAMIC ALLOCATION allows variation in array size.

You are responsible for creation AND deletion (No garbage collection in C++)

`int *pcppArray;`	`int *pcArray;`

Arrays and Pointers and Dynamic Allocation

DYNAMIC ALLOCATION allows variation in array size.

You are responsible for creation AND deletion (No garbage collection in C++)

```
int *pcppArray;

pcppArray = new int[5];
```

```
int *pcArray;

pcArray =
    (int *) calloc (5,sizeof(int));
```

Arrays and Pointers and Dynamic Allocation

DYNAMIC ALLOCATION allows variation in array size.

You are responsible for creation AND deletion (No garbage collection in C++)

```
int *pcppArray;

pcppArray = new int[5];


pcppArray[0] = 5;

*(++pcppArray) = 10;
```

```
int *pcArray;

pcArray =
   (int *) calloc (5,sizeof(int));

pcArray[0] = 5;

pcArray[1] = 10;
```

Arrays and Pointers and Dynamic Allocation

DYNAMIC ALLOCATION allows variation in array size.

You are responsible for creation AND deletion (No garbage collection in C++)

```
int *pcppArray;

pcppArray = new int[5];


pcppArray[0] = 5;

*(++pcppArray) = 10;

...

delete[] pcppArray;
```

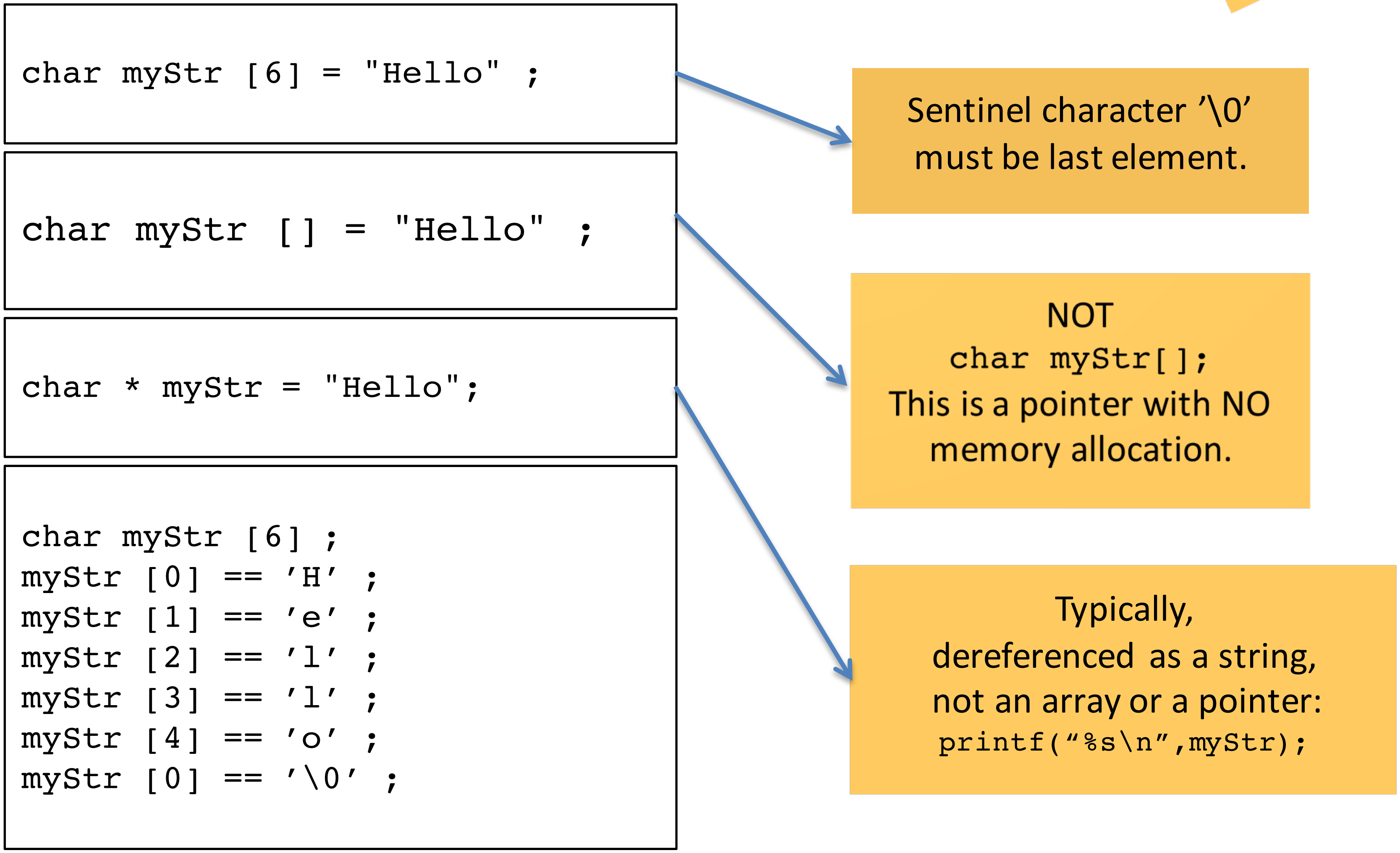
C++ has a
Vector container.
Use template:
#include <vector>

```
int *pcArray;

pcArray =
    (int *) calloc (5,sizeof(int));

pcArray[0] = 5;

pcArray[1] = 10;

...

free(pcArray);
```

Strings and Pointers

In C, strings are defined as arrays of *char*,
(i.e. a pointer is involved).

This is NOT
the C++ *string* type
found in
#include <string>



```
char myStr [6] = "Hello" ;
```

```
char myStr [] = "Hello" ;
```

```
char * myStr = "Hello";
```

```
char myStr [6] ;
myStr [0] == ’H’ ;
myStr [1] == ’e’ ;
myStr [2] == ’l’ ;
myStr [3] == ’l’ ;
myStr [4] == ’o’ ;
myStr [0] == ’\0’ ;
```

Sentinel character ’\0’
must be last element.

NOT
`char myStr[];`
This is a pointer with NO
memory allocation.

Typically,
dereferenced as a string,
not an array or a pointer:
`printf(“%s\n”,myStr);`

Odds and Ends

```
int myFunc(int x);

int *funcRetPtr(void);

void funcPtrArg(int (*inFunc)(int),char);

int main() {

    int (*foo)(int);
    foo = &myFunc;
    x = foo(150);
```

Pointers can be of type *function*.

*www.cprogramming.com/tutorial/**function-pointers**.html*

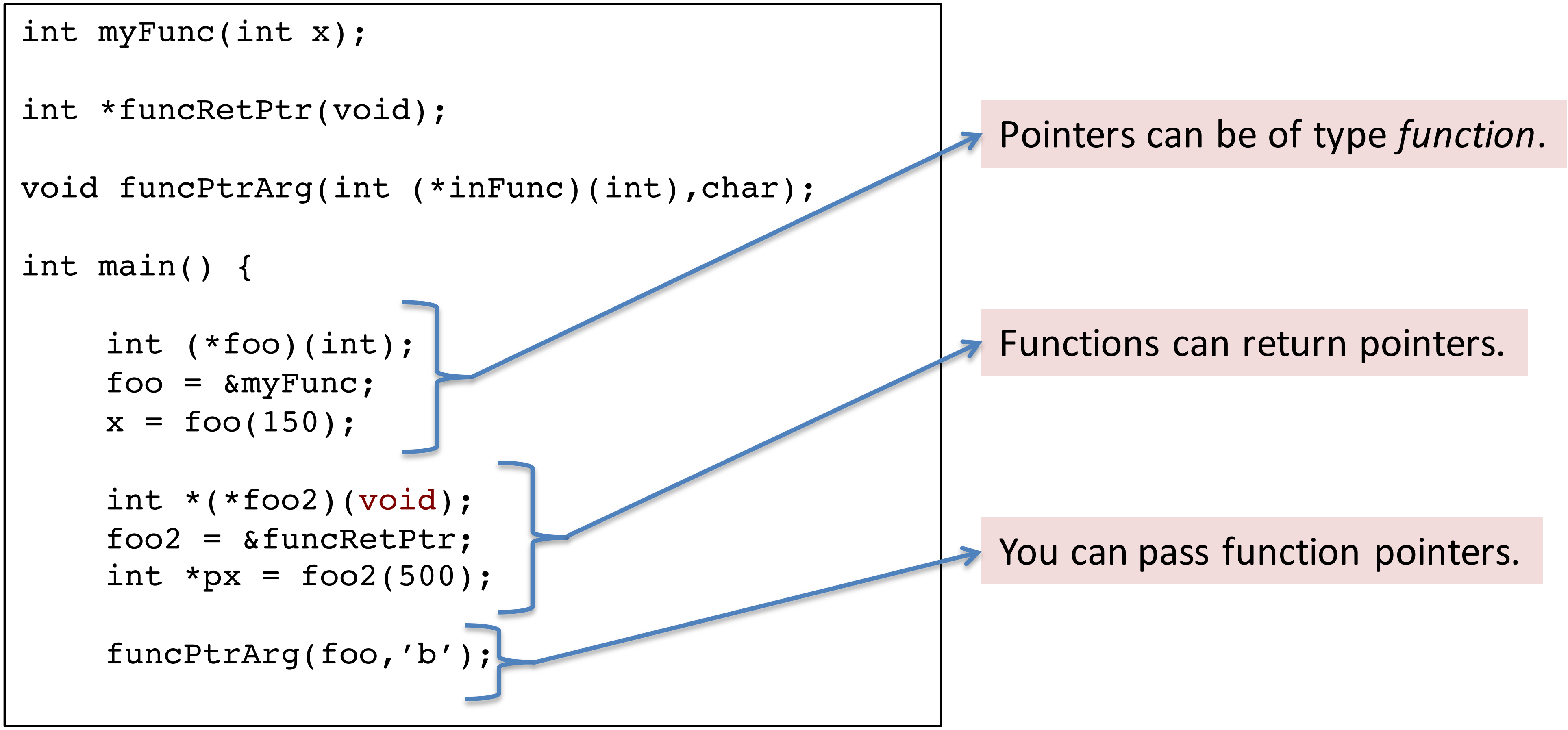
Odds and Ends

```
int myFunc(int x);

int *funcRetPtr(void);

void funcPtrArg(int (*inFunc)(int),char);

int main() {

    int (*foo)(int);
    foo = &myFunc;
    x = foo(150);

    int *(*foo2)(void);
    foo2 = &funcRetPtr;
    int *px = foo2(500);
```

Pointers can be of type *function*.

Functions can return pointers.

Odds and Ends



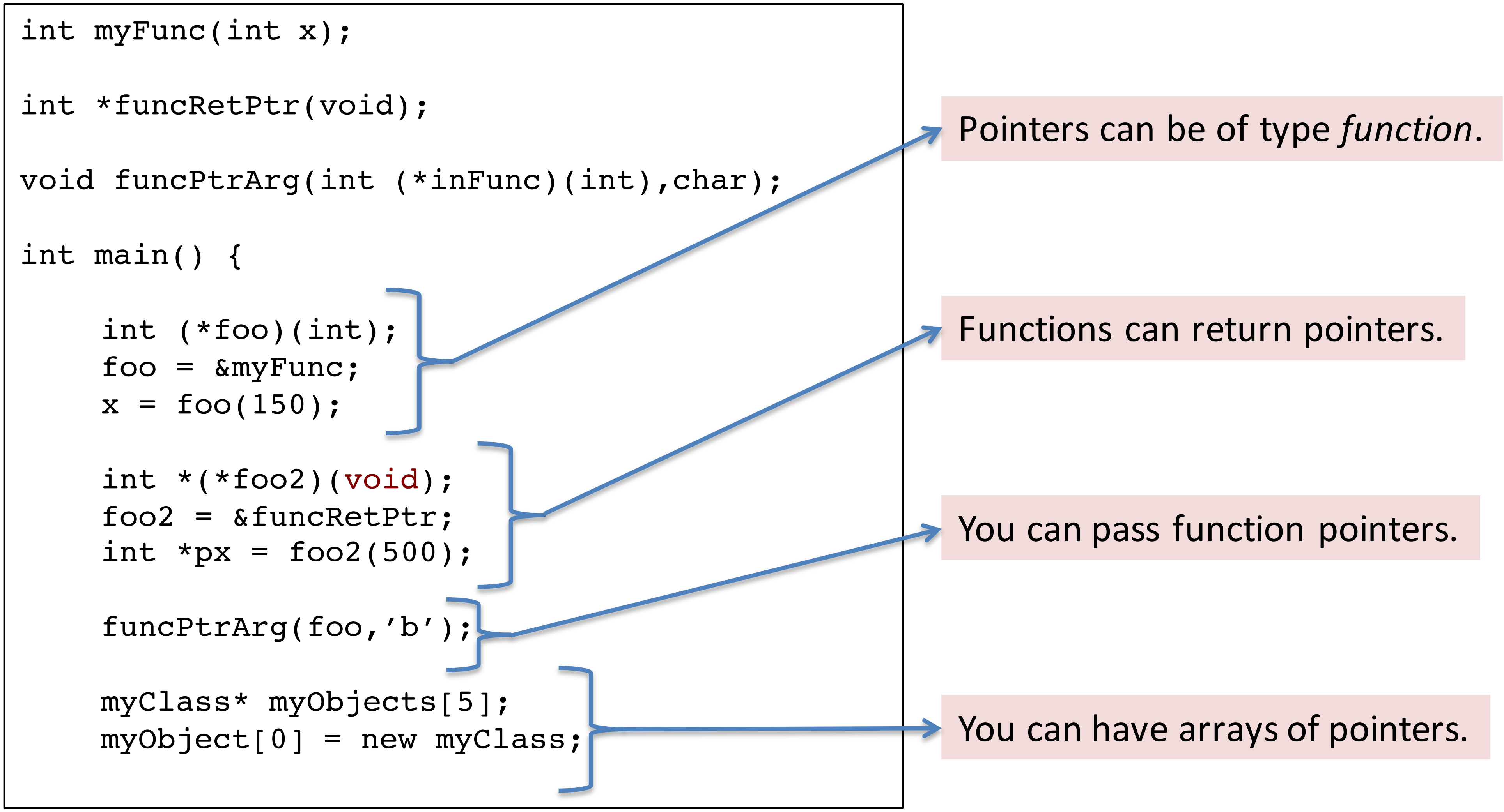
```
int myFunc(int x);

int *funcRetPtr(void);

void funcPtrArg(int (*inFunc)(int),char);

int main() {

     int (*foo)(int);
     foo = &myFunc;
     x = foo(150);

     int *(*foo2)(void);
     foo2 = &funcRetPtr;
     int *px = foo2(500);

     funcPtrArg(foo,’b’);
```

Pointers can be of type *function*.

Functions can return pointers.

You can pass function pointers.

Odds and Ends



```
int myFunc(int x);

int *funcRetPtr(void);

void funcPtrArg(int (*inFunc)(int),char);

int main() {

     int (*foo)(int);
     foo = &myFunc;
     x = foo(150);

     int *(*foo2)(void);
     foo2 = &funcRetPtr;
     int *px = foo2(500);

     funcPtrArg(foo,’b’);

     myClass* myObjects[5];
     myObject[0] = new myClass;
```

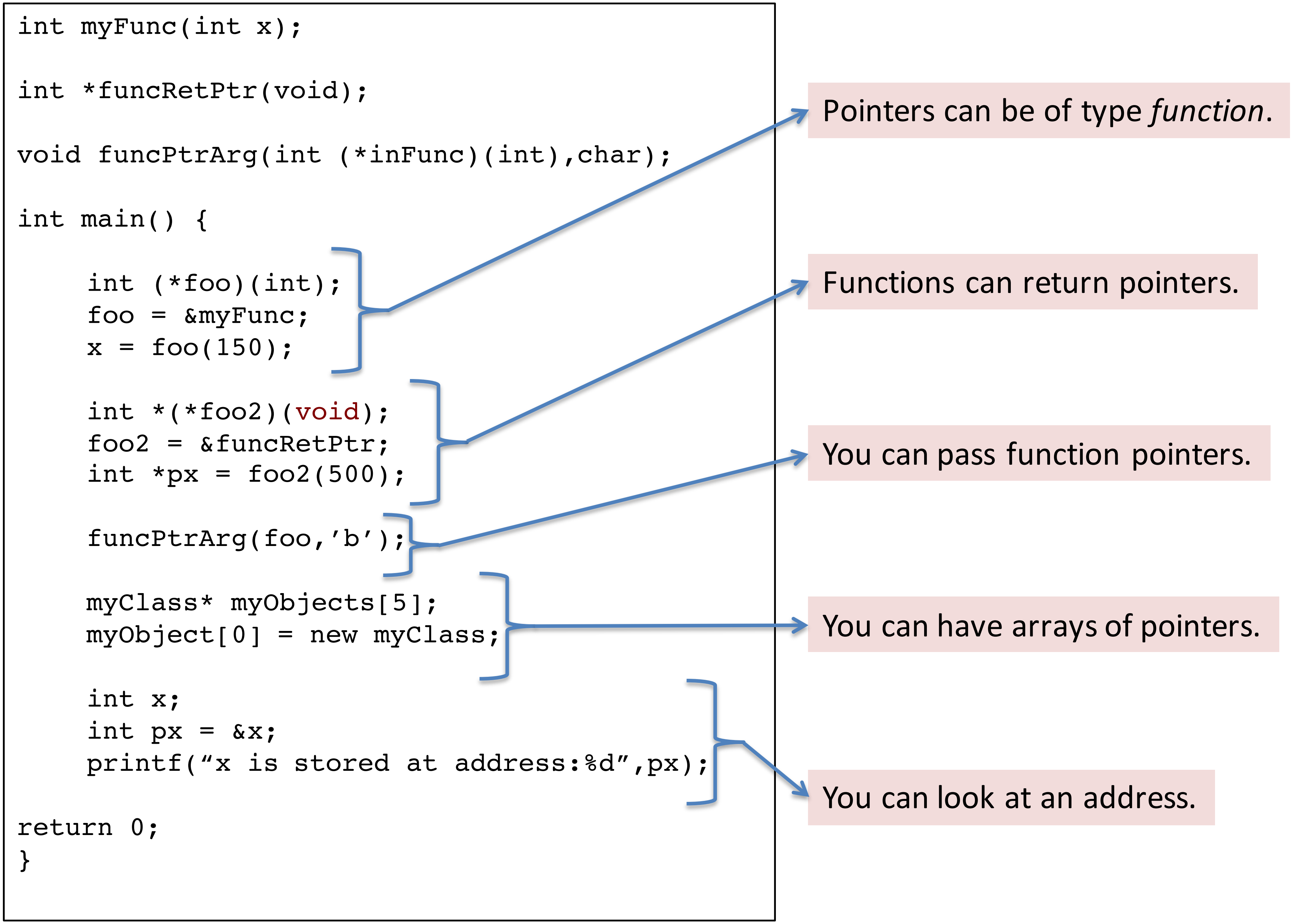
Pointers can be of type *function*.

Functions can return pointers.

You can pass function pointers.

You can have arrays of pointers.

Odds and Ends



```
int myFunc(int x);

int *funcRetPtr(void);

void funcPtrArg(int (*inFunc)(int),char);

int main() {

     int (*foo)(int);
     foo = &myFunc;
     x = foo(150);

     int *(*foo2)(void);
     foo2 = &funcRetPtr;
     int *px = foo2(500);

     funcPtrArg(foo,’b’);

     myClass* myObjects[5];
     myObject[0] = new myClass;

     int x;
     int px = &x;
     printf(“x is stored at address:%d”,px);

return 0;
}
```

Pointers can be of type *function*.

Functions can return pointers.

You can pass function pointers.

You can have arrays of pointers.

You can look at an address.

Pointers VERSUS References

Pointers (*)	References (&)
• Legal in C and C++	• Nonexistant in C
• Can be NULL and void!	• Must be initialized at declaration. • Must have a type.
• Very flexible and Dangerous.	• Constant and safe.
• Generally obvious.	• Hidden sometimes.
• Requires extra care with * &	• Eliminates confusing * &
• Good when you need it!	• Good for operator overloading.