Named Constants in C++

- Because it cannot be assigned to, a constant must be initialized.
- Constant can be evaluated at compile time.
- No store needs to be allocated for a constant because the compiler knows its value (depending on how smart compiler is, of course).
- It is typically necessary to allocate store for an array of constants because the compiler cannot in general figure out which elements of the array are referred to in expressions.

Named Constants (2)

```
int x = 1;
const int i1 = 2;
i1 = 3;         // error
i1++;           // error

const int i2 = x;
i2 = 4;         // error

const int v[] = {1, 2, 3, 4};
```

Named Constants (3)

```
x = 2;
cout << i2 << endl;          // print 1

const int *p1 = &i1;
cout << *p1 << endl;         // print 2

p1 = &i2;
cout << *p1 << endl;         // print 1
```

Named Constants (4)

```
p1 = &x;
cout << *p1 << endl;        // print 2

*p1 = 3;                    // error

int *p2 = &i1;              // error
```

Named Constants (5)

```
const int &i3 = x;
x = 4;
cout << i3 << endl;          // print 4

i3 = 5;                      // error

const int &i4 = i1;
cout << i4 << endl;          // print 2
```

Constant Objects

- Prefixing a declaration of a pointer with const makes the object, but not the pointer, a constant.

```
const char *pc = "asdf";
        // pointer to constant
pc[3] = 'a';        // error
pc = "ghjk";
```

Constant Pointers

- To declare a pointer itself, rather than the object pointed to, to be a constant, the operator ***constant** is used.

```
char *const cp = "asdf";  // constant pointer
cp[3] = 'a';              // ok?? ** error **
cp = "ghjk";              // error

char *pp = "asdf";
cout << pp[3] << endl;
pp[3] = 'a';              // ok?? ** error **
```

Constant Pointers (cont)

```
char qq[] = "asdf";
qq[3] = 'a';

int *const p3 = &x;
*p3 = 7;
p3 = &i1;                    // error

int *const p4 = &i1;         // error
```

Constant Objects & Pointers

- To make both object and pointer constant both must be declared const.

```
const char *const cpc = "asdf";
        // const pointer to const
cpc[3] = 'a';     // error
cpc = "ghjk";  // error
```

“const” in Arguments

- By declaring a pointer argument const, the function is prohibited from modifying the object pointed to.

char *strcpy(char *p, const char *q);

// cannot modify *q

“const” in Arguments (cont)

```
void f1(const int i) { }
void f2(int i) { i++; }
void f3(const int *i) { }
void f4(int *i) { *i++; }

f1(x);
f2(i1);
f3(&x);
f4(&i1);      // error
```

Strings in C++

char s[] = "ab"; char *s1 = s;

cout << *s1++ << ' ' << *s1++ << ' ‘

<< *s1++ << ' ' << endl;

// print “a b \0 \n“ 有可能印出 **\0 b a \n**

http://en.cppreference.com/w/cpp/language/eval_order

// "a" = 'a' + '\0'

cout << ('\0' == 0) << endl; // print 1

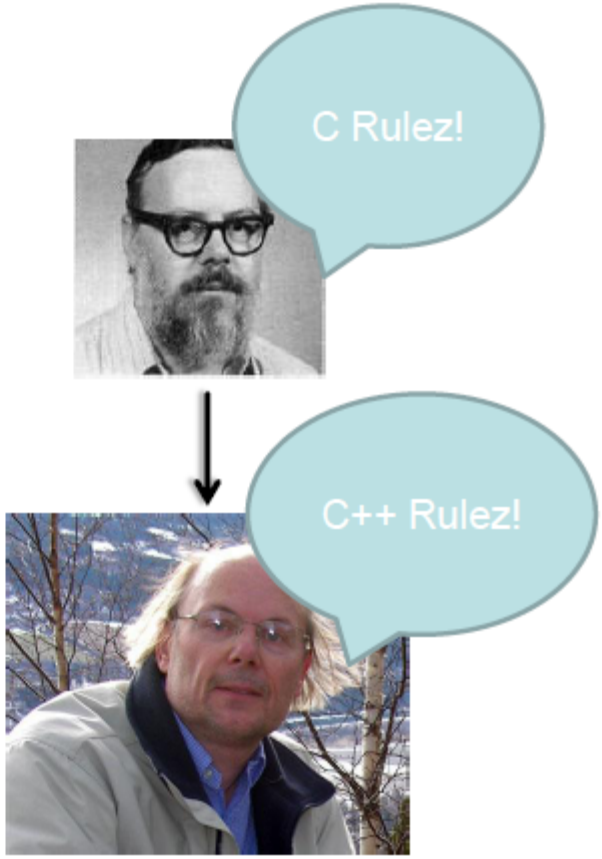
Strings in C++ (cont)

```
char ss1[MAX], ss2[MAX], *ss3;

for ( ; *ss1++; ) {
   *ss2++ = *ss1;     // copy ss1 to ss2
   *ss3++ = *ss1;     // error
}

*ss2 = '\0';       // absolutely necessary
```

This page intentionally left blank.

C++ Mini-Course

- Part 1: Mechanics
- Part 2: Basics
- Part 3: References
- Part 4: Const
- Part 5: Inheritance
- Part 6: Libraries



C++ Mini-Course

Part 1: Mechanics

C++ is a superset of C

- New Features include
  - Classes (Object Oriented)
  - Templates (Standard Template Library)
  - Operator Overloading
  - Slightly cleaner memory operations

Some C++ code

Segment.h

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};


#endif //   SEGMENT HEADER
```

Segment.cpp

```
#include "Segment.h"
#include "Point.h"

Segment::Segment()
{
    m_p0 = new Point(0, 0);
    m_p1 = new Point(1, 1);
}
Segment::~Segment()
{
    delete m_p0;
    delete m_p1;
}
```

#include

```
#include "Segment.h"
```

Insert header file at this point.

```
#include <iostream>
```

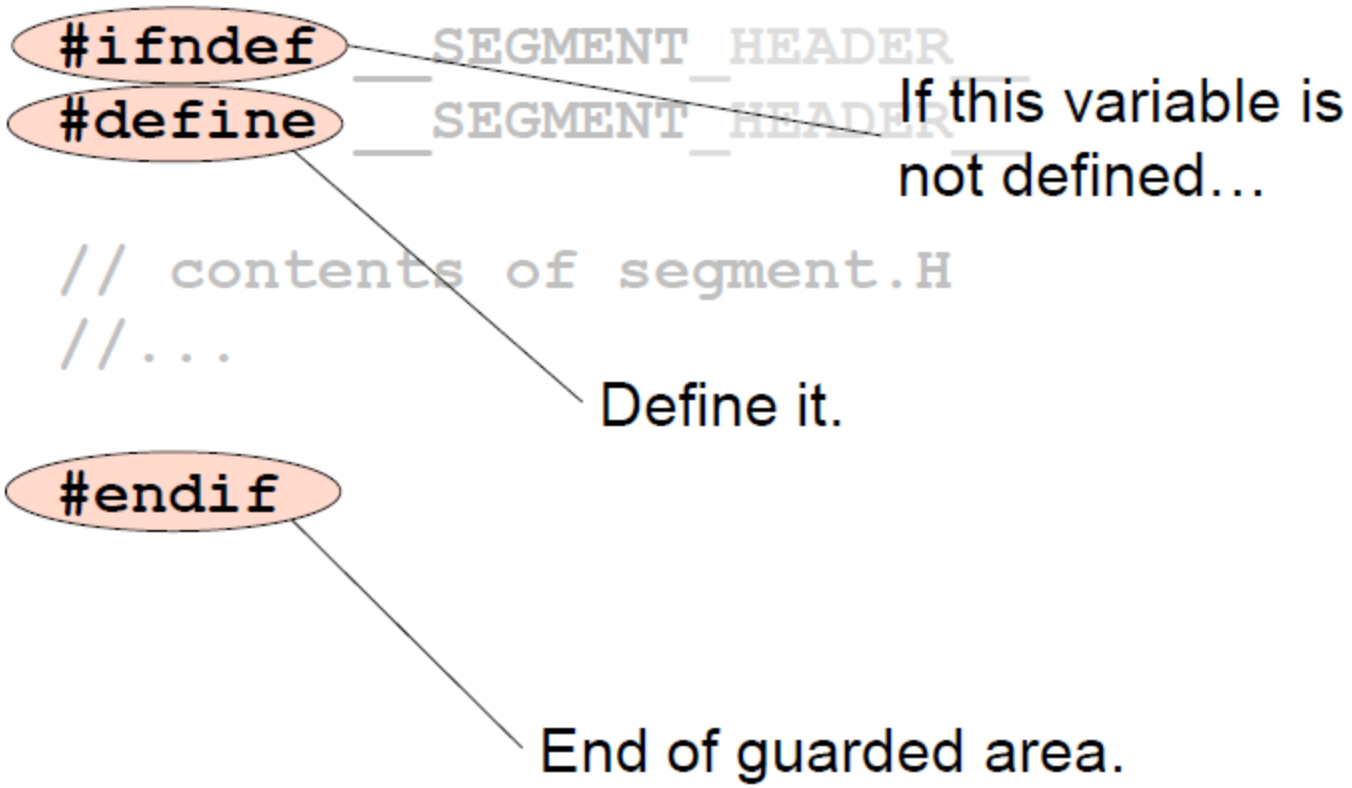
Use library header.

Header Guards

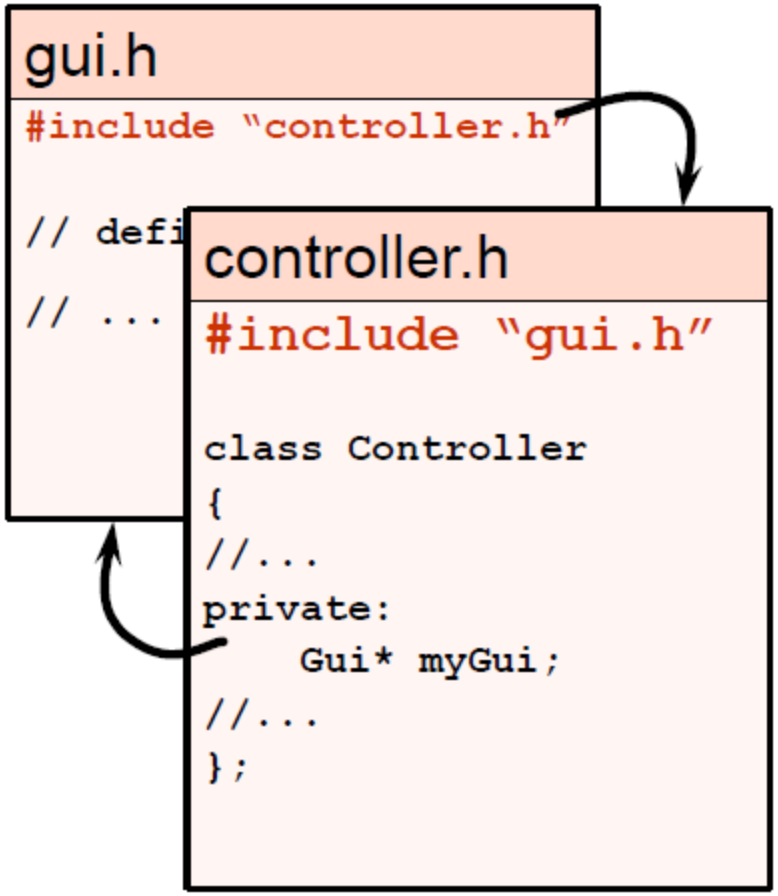
```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

// contents of Segment.h
//...

#endif
```

- To ensure it is safe to include a file more than once.

Header Guards

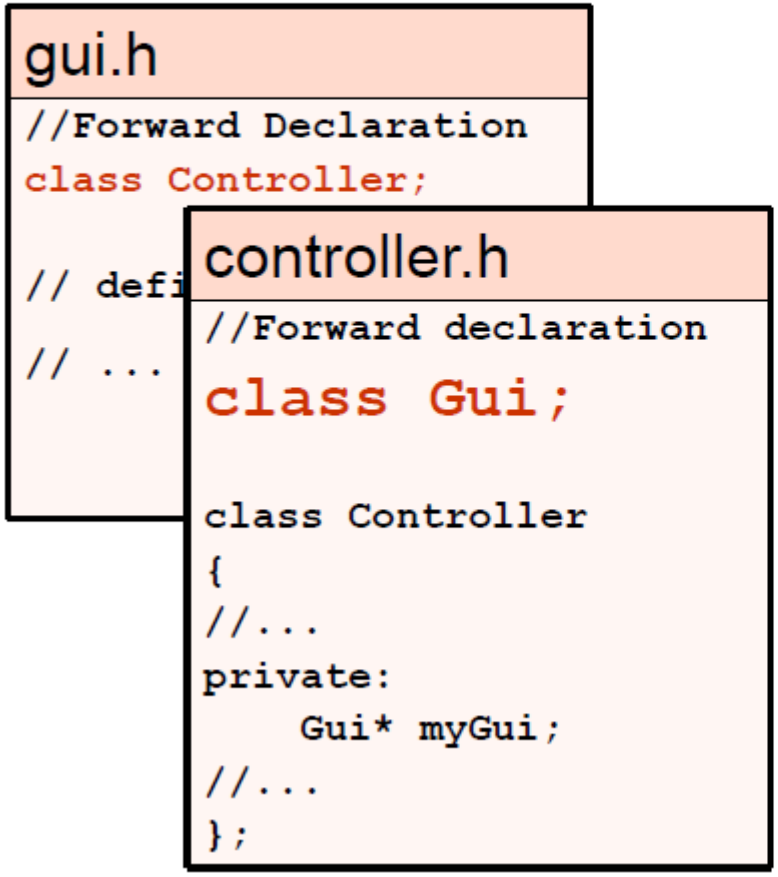


Circular Includes



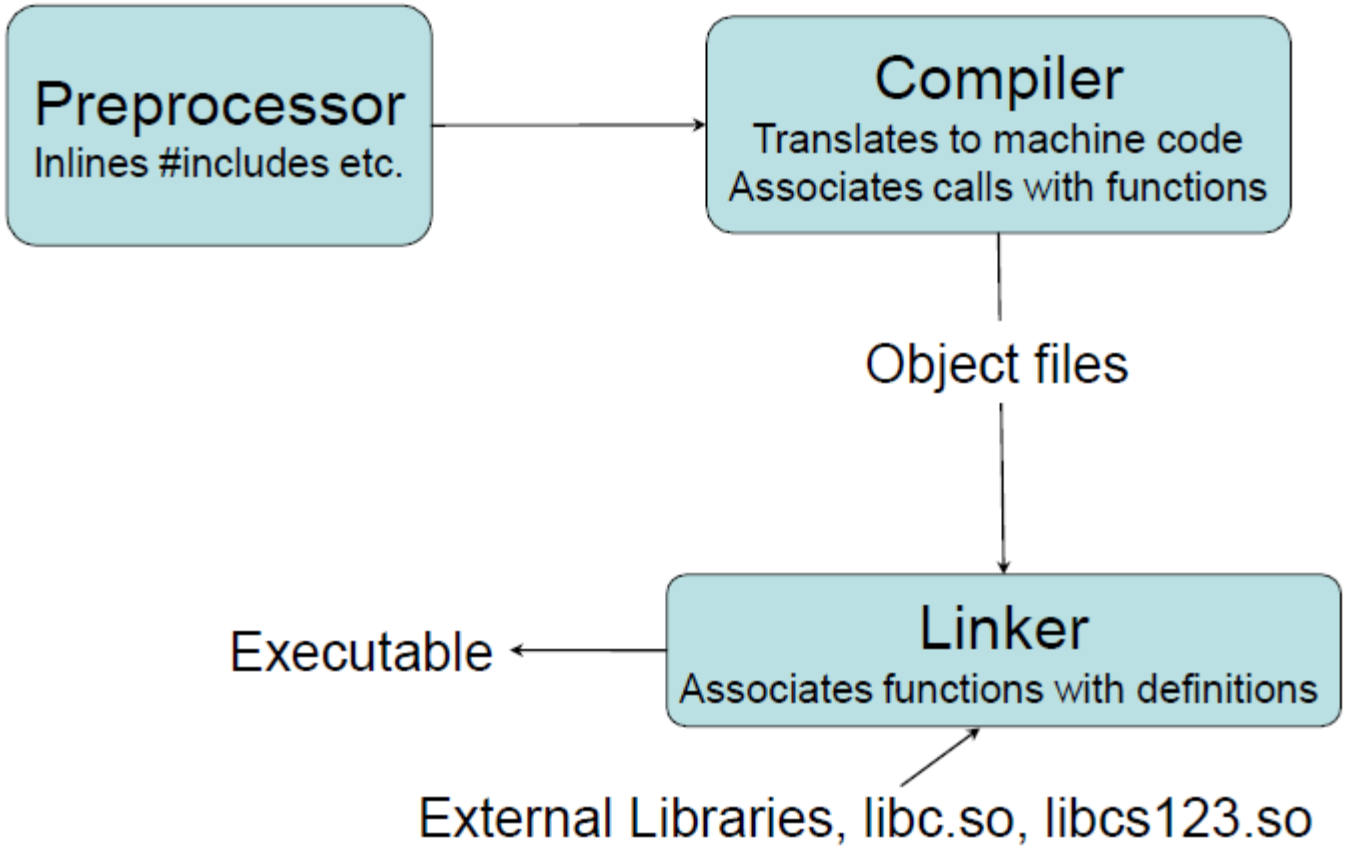
- What's wrong with this picture?
- How do we fix it?

Forward Declarations



- In header files, only include what you must.
- If only pointers to a class are used, use forward declarations.

Compilation



OK, OK. How do I run my Program?

```
> make
```

And if all goes well…

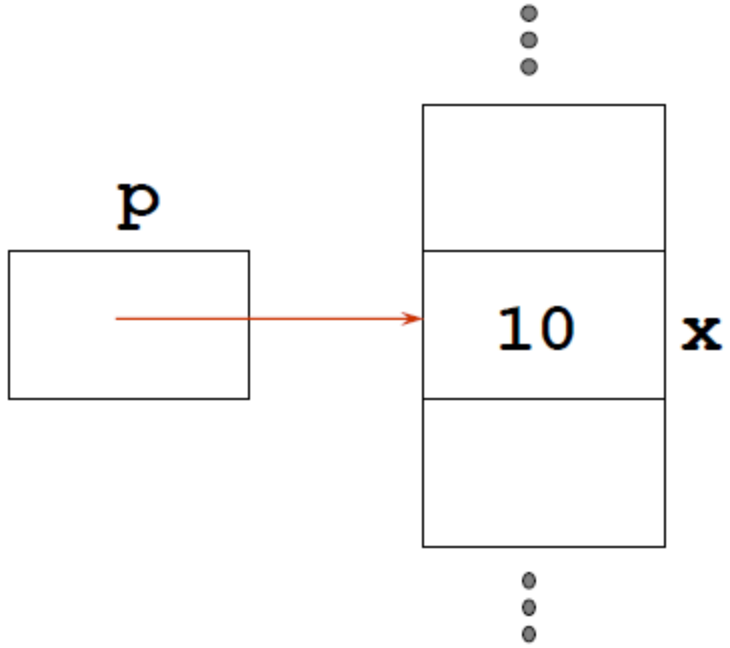
```
> ./myprog
```

C++ Mini-Course

Part 2: Basics

What is a pointer?

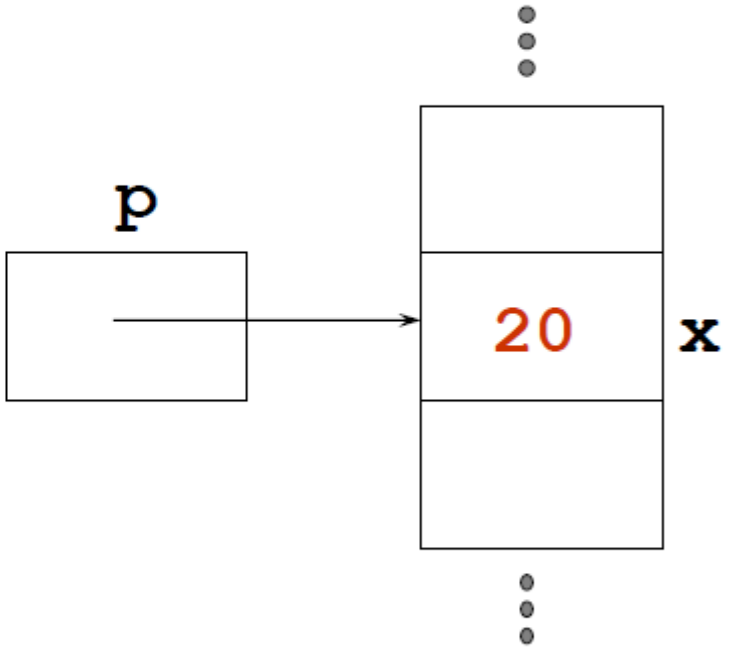
```
int x = 10;
int *p;

p = &x;
```



`p` gets the address of `x` in memory.

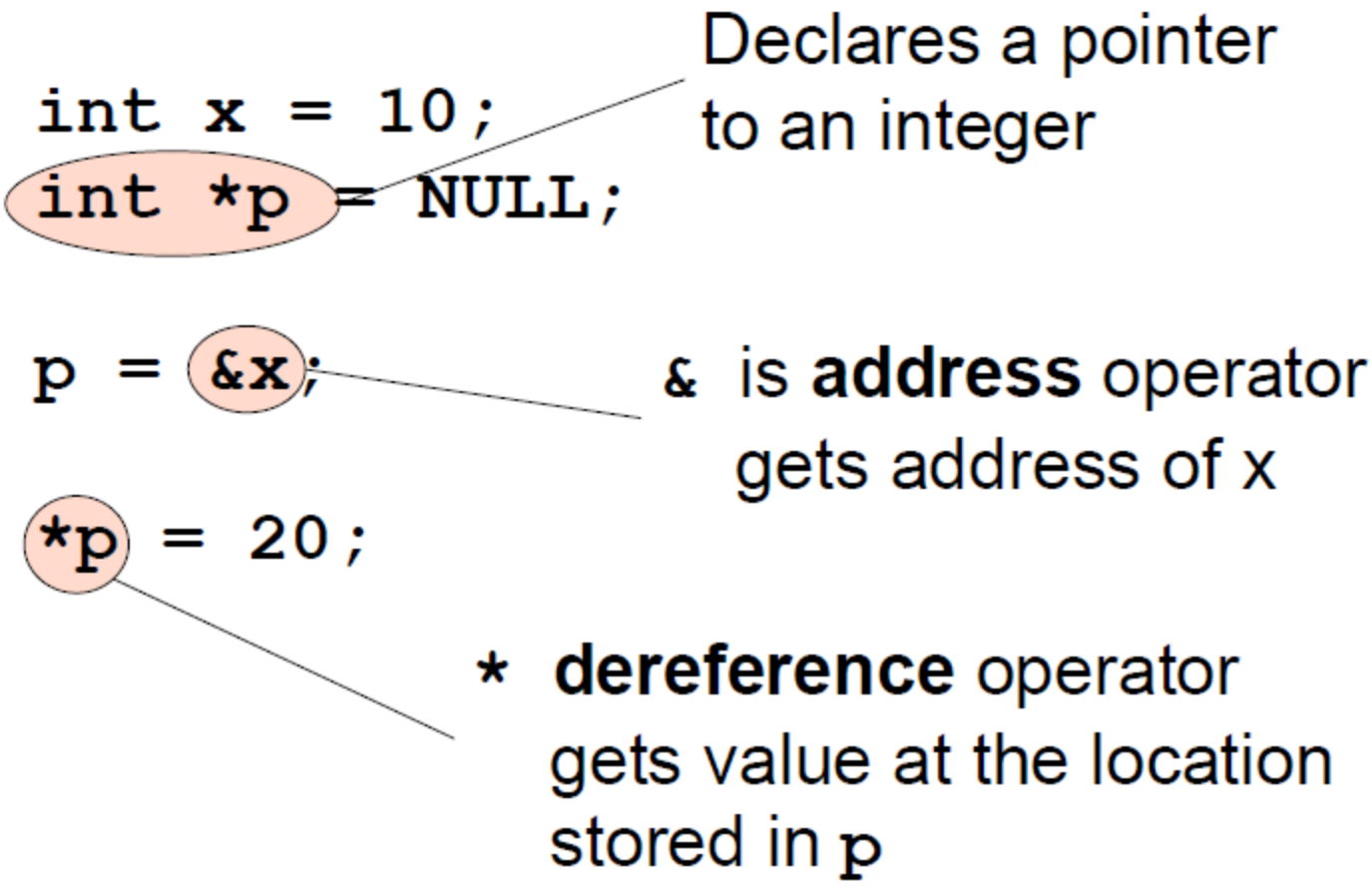
What is a pointer?

```
int x = 10;
int *p;

p = &x;

*p = 20;
```



`*p` is the value at the address `p`.

What is a pointer?



Allocating memory using **new**

```
Point *p = new Point(5, 5);
```

- **`new`** can be thought of as a function with slightly strange syntax
- **`new`** allocates space to hold the object.
- **`new`** calls the object's constructor.
- **`new`** returns a pointer to that object.

Deallocating memory using **delete**

```
// allocate memory
Point *p = new Point(5, 5);
...
// free the memory
delete p;
p = NULL;
```

For every call to **new**, there must be exactly one call to **delete**. It's a good practice to set to **NULL** afterwards to protect against double deletes.

Using **new** with arrays

```
int x = 10;
int* nums1 = new int[10];  // ok
int* nums2 = new int[x];   // ok
```

- Initializes an array of 10 integers on the heap.
- Equivalent to the following C code

```
int* nums = (int*)malloc(x * sizeof(int));
```

- Equivalent to the following Java code

```
int[] nums = new int[x];
```

Using **new** with multidimensional arrays

```
int x = 3, y = 4;
int** nums3 = new int[x][4];// ok
int** nums4 = new int[x][y];// BAD!
```

- Initializes a multidimensional array
- Only the first dimension can be a variable. The rest must be constants.
- Use single dimension arrays to fake multidimensional ones

Using **delete** on arrays

```
// allocate memory
int* nums1 = new int[10];
int* nums3 = new int[x][4][5];


...
// free the memory
delete[] nums1;
delete[] nums3;
```

- Have to use `delete[]`, or else only the first element is deleted.

Destructors

- `delete` calls the object's **destructor**.
- `delete` frees space occupied by the object.

- A **destructor** cleans up after the object.
- Releases resources such as memory.

Destructors – an Example

```
class Segment
{
public:
     Segment();
     virtual ~Segment();
private:
     Point *m_p0, *m_p1;
};
```

Destructors – an Example

```
Segment::Segment()
{
    m_p0 = new Point(0, 0);
    m_p1 = new Point(1, 1);
}
Segment::~Segment()
{
    delete m_p0;
    delete m_p1;
}
```

New vs Malloc

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

Malloc	New
Standard C Function	Operator (like ==, +=, etc.)
Used sparingly in C++; used frequently in C	Only in C++
Used for allocating chunks of memory of a given size without respect to what will be stored in that memory	Used to allocate instances of classes / structs / arrays and will invoke an object's constructor
Returns void* and requires explicit casting	Returns the proper type
Returns NULL when there is not enough memory	Throws an exception when there is not enough memory
Every malloc() should be matched with a free()	Every new/new[] should be matched with a delete/delete[]

Syntactic Sugar “->”

```
Point *p = new Point(5, 5);

// Access a member function:
(*p).move(10, 10);

// Or more simply:
p->move(10, 10);
```

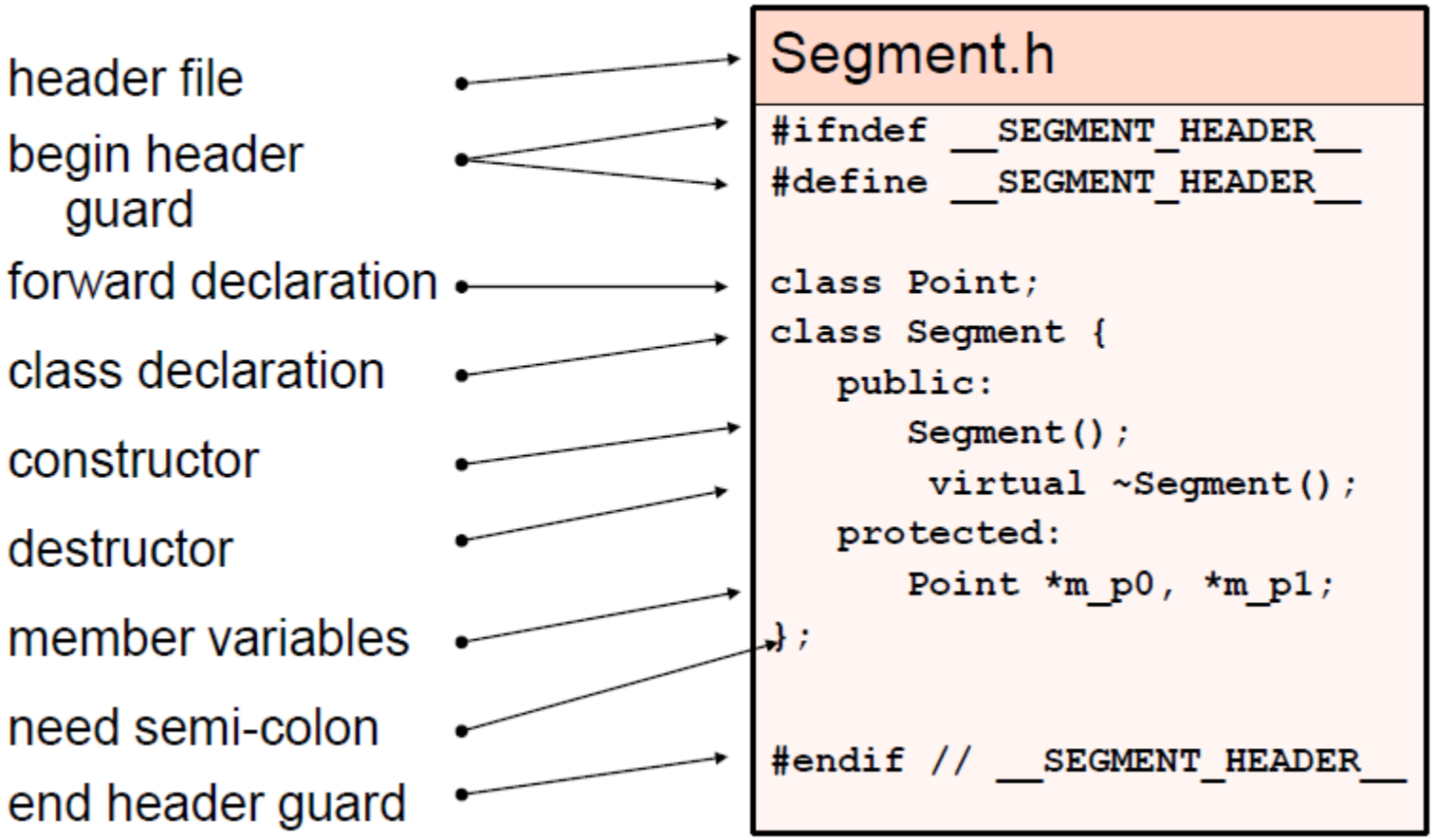
Stack vs. Heap

On the Heap / Dynamic allocation	On the Stack / Automatic allocation

`drawStuff() {`
`  Point *p = new Point();`
`  p->move(10,10);`
`  //...`
`}`
	`drawStuff() {`
	`  Point p();`
	`  p.move(5,5);`
	`  //...`
	`}`

What happens when `p` goes out of scope?

Summary with Header File



Syntax Note

- The following two statements mean the same thing
- They both allocate space for a Point object on the stack:
  - `Point p();`
  - `Point p = Point();`

C++ Mini-Course

Part 3: References

Passing by value

```
void Math::square(int i) {
     i = i*i;
}

int main() {
     int i = 5;
     Math::square(i);
     cout << i << endl;
}
```

Passing by reference

```
void Math::square(int &i) {
    i = i*i;
}

int main() {
    int i = 5;
    Math::square(i);
    cout << i << endl;
}
```

What is a reference?

- An alias – another name for an object.

```
int x = 5;
int &y = x; // y is a
            // reference to x
y = 10;
```

- What happened to x?
- What happened to y?

What is a reference?

- An alias – another name for an object.

```
int x = 5;
int &y = x; // y is a
            // reference to x
y = 10;
```

- What happened to x?
- What happened to y? – y **is** x.

Why are they useful?

- Unless you know what you are doing, do not pass objects by value; either use a pointer or a reference.
- Some people find it easier to deal with references rather then pointers, but in the end there is really only a syntactic difference (neither of them pass by value).
- Can be used to return more than one value (pass multiple parameters by reference)

Passing by reference: the bottom line

- The syntax is as though the parameter was passed by value.
- But behind the scenes, C++ is just passing a pointer.
- The following two are basically the same thing:

```
void increment(int &i)
{
     i = i + 1;
}
...
     int i;
     increment(i);
```

```
void increment(int *i)
{
     *i = i + 1;
}
...
     int i;
     increment(&i);
```

How are references different from Pointers?

Reference	Pointer
`int &a;`	`int *a;`

`int a = 10;`
`int b = 20;`
`int &c = a;`
`c = b;`
	`int a = 10;`
	`int b = 20;`
	`int *c = &a;`
	`c = &b;`

Asterisks and Ampersands

- In a type declaration, '*' indicates that you are declaring a pointer type.
  - Otherwise '*' is a dereference operator—gets the actual object from a pointer to the object.
- In a type declaration, '&' indicates that you are declaring a reference.
  - Otherwise '&' is the "address of" operator—gets a pointer to an object from the object itself.

C++ Mini-Course

Part 4: const

Introducing: `const`

```
void Math::printSquare(const int &i){
    i = i*i;
    cout << i << endl;
}

int main() {
    int i = 5;
    Math::printSquare(i);
    Math::printCube(i);
}
```

Won't compile.

Can also pass pointers to `const`

```
void Math::printSquare(const int *pi) {
    *pi = (*pi) * (*pi);
    cout << pi << endl;
}

int main() {
    int i = 5;
    Math::printSquare(&i);
    Math::printCube(&i);
}
```

← Still won't compile.

Declaring things `const`

```
const River nile;

const River* nilePc;

River* const nileCp;

const River* const nileCpc
```

Read pointer declarations right to left

```
// A const River
const River nile;

// A pointer to a const River
const River* nilePc;

// A const pointer to a River
River* const nileCp;

// A const pointer to a const River
const River* const nileCpc
```

Let's Try References

```
River nile;

const River &nileC = nile;

// Will this work?
River &nile1 = nileC;
```

How does `const` work here?

```
void Math::printSquares(const int &j,
  int &k) {
    k = k*k;  // Does this compile?
    cout << j*j << ", " << k << endl;
}

int main() {
    int i = 5;
    Math::printSquares(i, i);
}
```

Returning `const` references is OK

```
class Point {
  public:
    const double &getX() const;
    const double &getY() const;
    void move(double dx, double dy);

  protected:
    double m_x, m_y;
};
```

```
const double &
Point::getX() const {
    return m_x;
}
```

Function won't change `*this`.

C++ Mini-Course

Part 5: Inheritance

Classes vs Structs

- Default access specifier for classes is private; for structs it is public
- Except for this difference, structs are functionally the same as classes, but the two are typically used differently: structs should be thought of as lightweight classes that contain mostly data and possibly convenience methods to manipulate that data and are hardly ever used polymorphically

```
struct Point {
    int x;
    int y;

    // convenience constructor
    Point(int a, int b)
       : x(a), y(b)
    { }

    // @returns distance to another point
    double distance(const Point &pnt) {
        int dx = m_x - pnt.x;
        int dy = m_y - pnt.y;
        return math.sqrt(dx*dx + dy*dy);
    }
};
```

```
class Segment {
   public:
        Segment();
        virtual ~Segment();

        void setPoints(int x0, int y0, int x1, int y1);

    protected:
        Point *m_p0, *m_p1;
};

void Segment::setPoints(int x0, int y0, int x1, int y1) {
    m_p0 = new Point(x0, y0);
    m_p1 = new Point(x1, y1);
}
```

How does inheritance work?

must include parent header file

DottedSegment publicly inherits from Segment

```
#include "Segment.h"
class DottedSegment : public Segment
{
        // DottedSegment declaration
};
```

virtual

- In Java every method invocation is dynamically bound, meaning for every method invocation the program checks if a sub-class has overridden the method. You can disable this (somewhat) by using the keyword “final” in Java
- In C++ you have to declare the method virtual if you want this functionality. (So, “virtual” is the same thing as “not final”)
- You should declare methods virtual when they are designed to be overridden or will otherwise participate in an inheritance hierarchy.

pure virtual functions

- In Java, the “abstract” keyword means the function is undefined in the superclass.
- In C++, we use pure virtual functions:
  - virtual int mustRedfineMe(char *str) = 0;
  - This function must be implemented in a subclass.

Resolving functions

In Java:

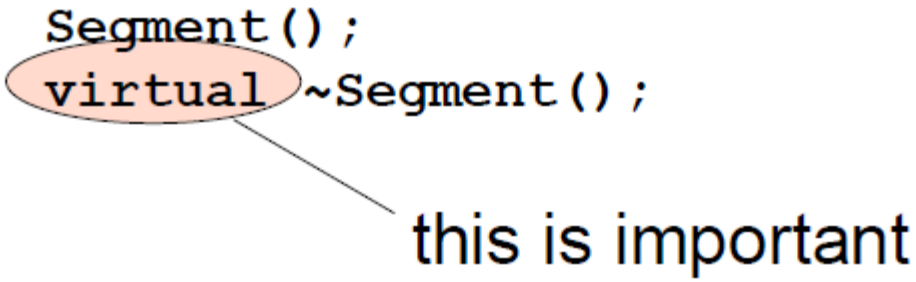
```
// Overriding methods
public void overloaded(){
  println("woohoo");
  super.overloaded();
}

//constructor
public Subclass(){
  super();
}
```

In C++:

```
// Overriding methods
void Subclass::overloaded(){
  cout<<"woohoo"<<endl;
  Superclass::overloaded();
}

//constructor
public Subclass() :
  Superclass()
{ }
```

Make destructors virtual

- Make sure you declare your destructors virtual; if you do not declare a destructor a non-virtual one will be defined for you



C++ Mini-Course

Part 6: Libraries

Namespaces

- Namespaces are kind of like packages in Java
- Reduces naming conflicts
- Most standard C++ routines and classes and under the `std` namespace
  - Any standard C routines (malloc, printf, etc.) are defined in the global namespace because C doesn't have namespaces

using namespace

```
#include <iostream>
...
std::string question =
            "How do I prevent RSI?";
std::cout << question << std::endl;

using namespace std;

string answer = "Type less.";
cout << answer << endl;
```

Bad practice to do in header files!

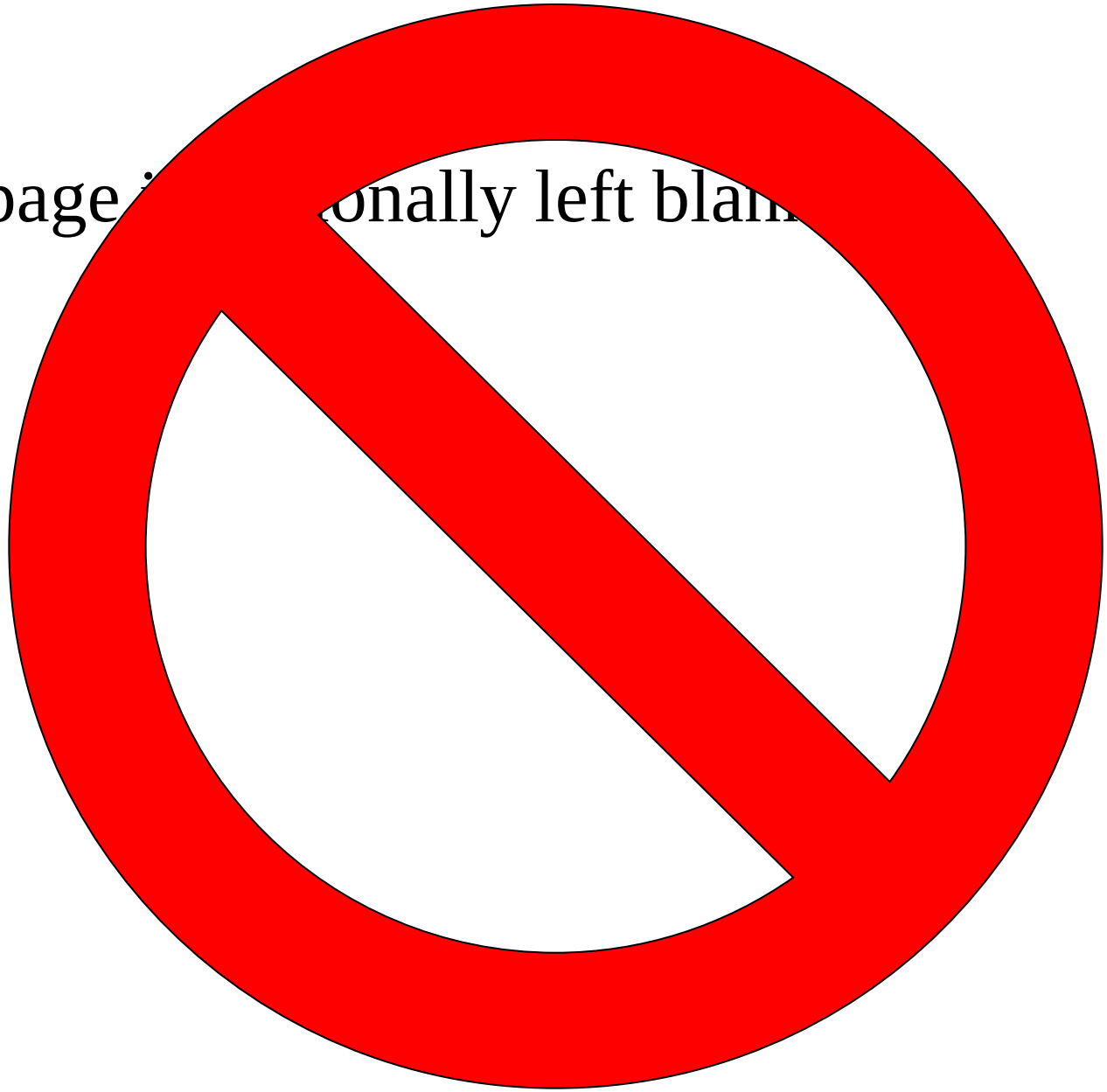
STL

- Standard Template Library
- Contains well-written, templated implementations of most data structures and algorithms
  - Templates are similar to generics in Java
  - Allows you to easily store anything without writing a container yourself
- Will give you the most hideous compile errors ever if you use them even slightly incorrectly!

STL example

```
#include <vector>
using namespace std;

typedef vector<Point> PointVector;
typedef PointVector::iterator PointVectorIter;

PointVector v;
v.push_back(Point(3, 5));

PointVectorIter iter;
for(iter = v.begin(); iter != v.end(); ++iter){
   Point &curPoint = *iter;
}
```

This page i onally left blan

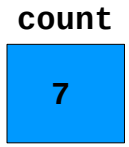


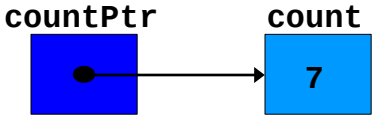
Chapter 5 - Pointers and Strings

Outline

5.2 Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as values
  - Normally, variable contains specific value (direct reference)
  - Pointers contain address of variable that has specific value (indirect reference)





5.2 Pointer Variable Declarations and Initialization

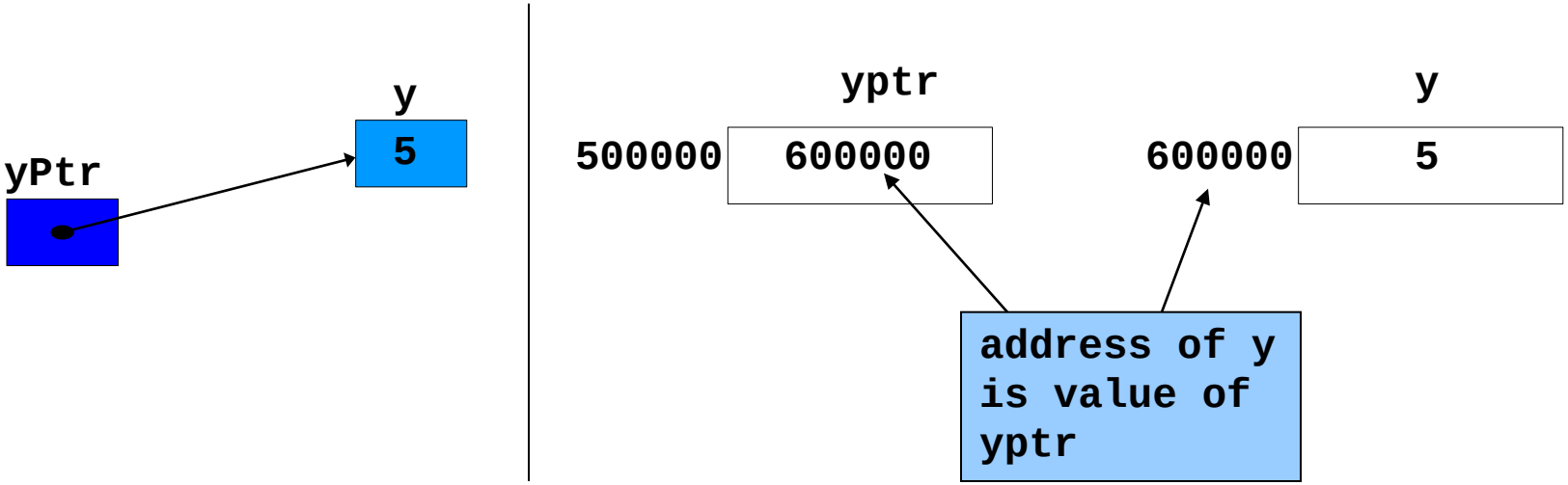
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **`0`**, **`NULL`**, or address
    - **`0`** or **`NULL`** points to nothing

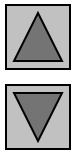
5.3 Pointer Operators

- **&** (address operator)
  - Returns memory address of its operand
  - Example

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

  - **yPtr** "points to" **y**





Outline

fig05_04.cpp (1 of 2)

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int a;      // a is an integer
11    int *aPtr;  // aPtr is a pointer to an integer
12
13    a = 7;
14    aPtr = &a;  // aPtr assigned address of a
15
16    cout << "The address of a is " << &a
17         << "\nThe value of aPtr is " << aPtr;
18
19    cout << "\n\nThe value of a is " << a
20         << "\nThe value of *aPtr is " << *aPtr;
21
22    cout << "\n\nShowing that * and & are inverses of "
23         << "each other.\n&*aPtr = " << &*aPtr
24         << "\n*&aPtr = " << *&aPtr << endl;
25
```

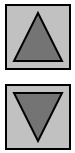
***** and **&** are inverses of each other

5.4 Calling Functions by Reference

- 3 ways to pass arguments to function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- `return` can return one value from function
- Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value "returned"

5.5 Using `const` with Pointers

- **`const`** pointers
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared



Outline

fig05_13.cpp (1 of 1)

fig05_13.cpp output (1 of 1)

```
1  // Fig. 5.13: fig05_13.cpp
2  // Attempting to modify a constant pointer to
3  // non-constant data.
4
5  int main()
6  {
7     int x, y;
8
9     // ptr is a constant pointer to an int
10    // be modified through pt                 the
11    // same memory location.
12    int * const ptr = &x;
13
14    *ptr = 7;  // allowed: *p
15    ptr = &y;  // error: ptr                  address
16
17    return 0;  // indicates successful ter
18
19 } // end main
```

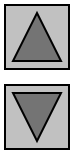
```
d:\cpphtp4_examples\ch05\Fig05_13.cpp(15) : e
   l-value specifies const object
```

ptr is constant pointer to integer.

Can modify **x** (pointed to by **ptr**) since x not constant.

Cannot modify **ptr** to point to new address since **ptr** is constant.

Line 15 generates compiler error by attempting to assign new address to constant pointer.



Outline

fig05_14.cpp
(1 of 1)

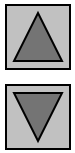
```
1  // Fig. 5.14: fig05_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int x = 5, y;
11
12    // ptr is a constant pointer to a co
13    // ptr always points to the same loc
14    // at that location cannot
15    const int *const ptr = &x
16
17    cout << *ptr << endl;
18
19    *ptr = 7;  // error: *ptr          value
20    ptr = &y;  // error: ptr           ddress
21
22    return 0;  // indicates successful termination
23
24 } // end main
```

ptr is constant pointer to integer constant.

Cannot modify **x** (pointed to by **ptr**) since ***ptr** declared constant.

Cannot modify **ptr** to point to new address since **ptr** is constant.

```
d:\cpphtp4_examples\ch05\Fig05_14.cpp(19) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch05\Fig05_14.cpp(20) : error C2166:
   l-value specifies const object
```



Outline

_14.cpp
ut (1 of 1)

Line 19 generates compiler error by attempting to modify constant object.

Line 20 generates compiler error by attempting to assign new address to constant pointer.

5.6 Bubble Sort Using Pass-by-Reference

- **`sizeof`**
  - Unary operator returns size of operand in bytes
  - For arrays, **`sizeof`** returns

    (size of 1 element) * (number of elements)
  - If **`sizeof( int ) = 4`**, then

```
int myArray[10];
cout << sizeof(myArray);
```

    will print 40
- **`sizeof`** can be used with
  - Variable names
  - Type names
  - Constant values

5.7 Pointer Expressions and Pointer Arithmetic

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

5.8 Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like constant pointer
  - Pointers can do array subscripting operations
- Accessing array elements with pointers
  - Element **`b[ n ]`** can be accessed by **`*( bPtr + n )`**
    - Called pointer/offset notation
  - Addresses
    - **`&b[ 3 ]`** same as **`bPtr + 3`**
  - Array name can be treated as pointer
    - **`b[ 3 ]`** same as **`*( b + 3 )`**
  - Pointers can be subscripted (pointer/subscript notation)
    - **`bPtr[ 3 ]`** same as **`b[ 3 ]`**

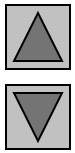
5.11 Function Pointers

- Pointers to functions
  - Contain address of function
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers

5.11 Function Pointers

- Calling functions using pointers
  - Assume parameter:
    - **`bool ( *compare ) ( int, int )`**
  - Execute function with either
    - **`( *compare ) ( int1, int2 )`**
      - Dereference pointer to function to execute

    OR
    - **`compare( int1, int2 )`**

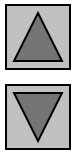


Outline

fig05_25.cpp
(1 of 5)

```
1  // Fig. 5.25: fig05_25.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 // prototypes
14 void bubble( int [], const int, bool (*)( int, int ) );
15 void swap( int * const, int * const );
16 bool ascending( int, int );
17 bool descending( int, int );
18
19 int main()
20 {
21    const int arraySize = 10;
22    int order;
23    int counter;
24    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
25
```

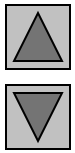
Parameter is pointer to function that receives two integer parameters and returns **`bool`** result.



Outline

fig05_25.cpp
(2 of 5)

```
26    cout << "Enter 1 to sort in ascending order,\n"
27         << "Enter 2 to sort in descending order: ";
28    cin >> order;
29    cout << "\nData items in original order\n";
30
31    // output original array
32    for ( counter = 0; counter < arraySize; counter++ )
33       cout << setw( 4 ) << a[ counter ];
34
35    // sort array in ascending order; pass function ascending
36    // as an argument to specify ascending sorting order
37    if ( order == 1 ) {
38       bubble( a, arraySize, ascending );
39       cout << "\nData items in ascending order\n";
40    }
41
42    // sort array in descending order; pass function descending
43    // as an agrument to specify descending sorting order
44    else {
45       bubble( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    }
48
```



Outline

fig05_25.cpp
(3 of 5)

```
49    // output sorted array
50    for ( counter = 0; counter < arraySize; counter++ )
51       cout << setw( 4 ) << a[ counter ];
52
53    cout << endl;
54
55    return 0;  // indicates successful termination
56
57 } // end main
58
59 // multipurpose bubble sort; parameter comp
60 // the comparison function that determines
61 void bubble( int work[], const int size,
62              bool (*compare)( int, int ) )
63 {
64    // loop to control passes
65    for ( int pass = 1; pass < size; pass++
66
67       // loop to control number of c
68       for ( int count = 0; count < s
69
70          // if adjacent elements are
71          if ( (*compare)( work[ count ], work[ count + 1 ] ) )
72             swap( &work[ count ], &work[ count + 1 ] );
```

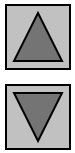
compare is pointer to function that receives two integer parameters and returns **bool** result.

Parentheses necessary to indicate pointer to function

Call passed function **compare**; dereference pointer to execute function.

5.11 Function Pointers

- Arrays of pointers to functions
  - Menu-driven systems
  - Pointers to each function stored in array of pointers to functions
    - All functions must have same return type and same parameter types
  - Menu choice → subscript into array of function pointers



Outline

fig05_26.cpp
(1 of 3)

```
1  // Fig. 5.26: fig05_26.cpp
2  // Demonstrating an array of pointers to functions.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // function prototypes
10 void function1( int );
11 void function2( int );
12 void function3( int );
13
14 int main()
15 {
16    // initialize array of 3 pointers to fu
17    // take an int argument and return void
18    void (*f[ 3 ])( int ) = { function1, function2, function3 };
19
20    int choice;
21
22    cout << "Enter a number between 0 and 2, 3 to end: ";
23    cin >> choice;
24
```

Array initialized with names of three functions; function names are pointers.

5.12.1 Fundamentals of Characters and Strings

- Character constant
  - Integer value represented as character in single quotes
- String
  - Array of characters, ends with null character `'\0'`
  - String is constant pointer
    - Pointer to string's first character
      - Like arrays

5.12.1 Fundamentals of Characters and Strings

- String assignment
  - Character array
    - **`char color[] = "blue";`**
      - Creates 5 element **`char`** array **`color`**
        - last element is **`'\0'`**
  - Variable of type **`char *`**
    - **`char *colorPtr = "blue";`**
      - Creates pointer **`colorPtr`** to letter **`b`** in string **`“blue”`**
        - **`“blue”`** somewhere in memory
  - Alternative for character array
    - **`char color[] = { ‘b’, ‘l’, ‘u’, ‘e’, ‘\0’ };`**

5.12.1 Fundamentals of Characters and Strings

- Reading strings
  - Assign input to character array **`word[ 20 ]`**

```
cin >> word
```

    - Reads characters until whitespace or EOF
    - String could exceed array size

```
cin >> setw( 20 ) >> word;
```

    - Reads 19 characters (space reserved for **`'\0'`**)

5.12.1 Fundamentals of Characters and Strings

- **`cin.getline`**
  - Read line of text
  - **`cin.getline( array, size, delimiter );`**
  - Copies input into specified **`array`** until either
    - One less than **`size`** is reached
    - **`delimiter`** character is input
  - Example

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```


5.12.2 String Manipulation Functions of the String-handling Library

`char *strcpy( char *s1, const char *s2 );`	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most **n** characters of string **s2** to string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string **s1** with the string **s2**. The function returns a value of zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.

5.12.2 String Manipulation Functions of the String-handling Library

`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to **n** characters of the string **s1** with the string **s2**. The function returns zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—delimited by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current to-ken is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.
`size_t strlen( const char *s );`	Determines the length of string **s**. The number of characters preceding the terminating null character is returned.

5.12.2 String Manipulation Functions of the String-handling Library

- Copying strings
  - **`char *strcpy( char *s1, const char *s2 )`**
    - Copies second argument into first argument
      - First argument must be large enough to store string and terminating null character
  - **`char *strncpy( char *s1, const char *s2, size_t n )`**
    - Specifies number of characters to be copied from string into array
    - Does not necessarily copy terminating null character

5.12.2 String Manipulation Functions of the String-handling Library

- Concatenating strings
  - **`char *strcat( char *s1, const char *s2 )`**
    - Appends second argument to first argument
    - First character of second argument replaces null character terminating first argument
    - Ensure first argument large enough to store concatenated result and null character
  - **`char *strncat( char *s1, const char *s2, size_t n )`**
    - Appends specified number of characters from second argument to first argument
    - Appends terminating null character to result

5.12.2 String Manipulation Functions of the String-handling Library

- Comparing strings
  - **`int strcmp( const char *s1, const char *s2 )`**
    - Compares character by character
    - Returns
      - Zero if strings equal
      - Negative value if first string less than second string
      - Positive value if first string greater than second string
  - **`int strncmp( const char *s1, const char *s2, size_t n )`**
    - Compares up to specified number of characters
    - Stops comparing if reaches null character in one of arguments

5.12.2 String Manipulation Functions of the String-handling Library

- Tokenizing
  - Breaking strings into tokens, separated by delimiting characters
  - Tokens usually logical units, such as words (separated by spaces)
  - **`"This is my string"`** has 4 word tokens (separated by spaces)
  - **`char *strtok( char *s1, const char *s2 )`**
    - Multiple calls required
      - First call contains two arguments, string to be tokenized and string containing delimiting characters
        - Finds next delimiting character and replaces with null character
      - Subsequent calls continue tokenizing
        - Call with first argument **`NULL`**

5.12.2 String Manipulation Functions of the String-handling Library

- Determining string lengths
  - **`size_t strlen( const char *s )`**
    - Returns number of characters in string
      - Terminating null character not included in length