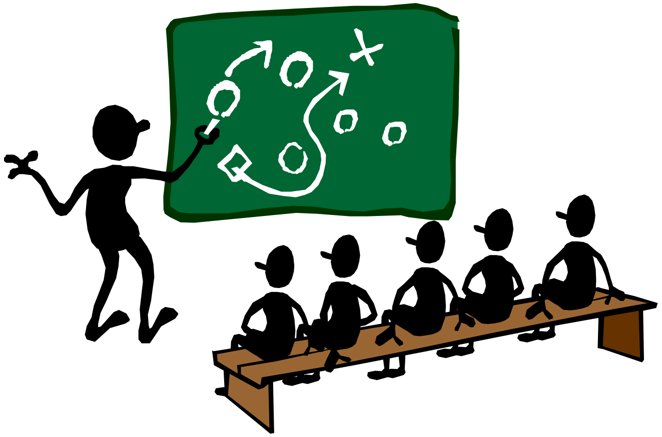
C++ Programming Language
Chapter 10
Pointer and Dynamic Arrays



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

April 2011

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Learning Objectives

- Pointers
  - pointer operations (address-of, deference, assignment)
  - pointer arithmetic
- Dynamic memory management
  - how to allocate(create), use, and deallocate(destroy)
  - 4 operators: new, new[], delete, and delete[]
- Classes, pointers, arrays, and dynamic memory
  - **this** pointer and destructors (dtors)
  - revisit ctor, copy ctor, assignment operator
  - dynamic arrays of class objects

Pointer Introduction

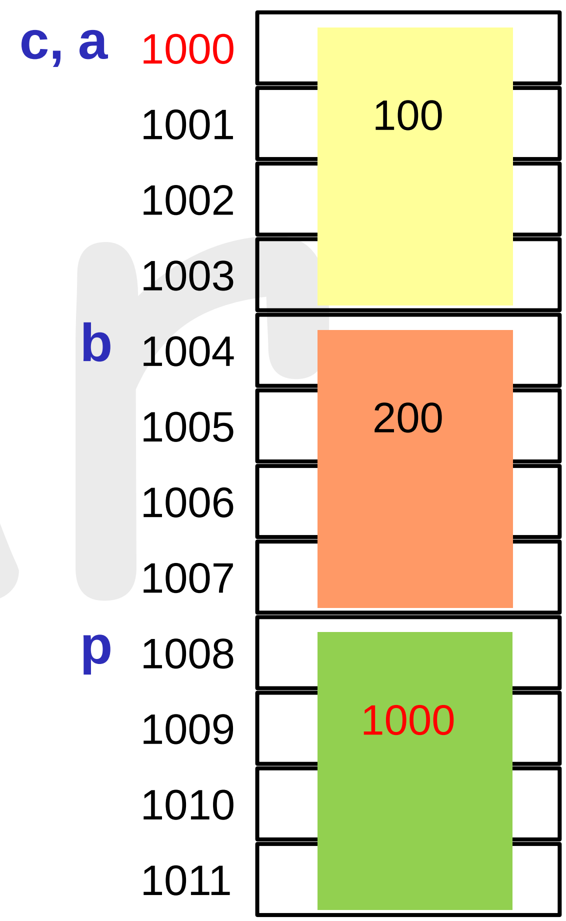
- Memory in C++
  - numbered memory locations
    - unique number for each byte
  - linear addressing
- Variable name is an alias of memory address

  int a, b; // assume sizeof(int) = 4

  a = 100; b = 200;

  int& c = a; // c is a reference (alias) of a
- Pointer variable
  - its value IS a memory address

  int *p = &c; // assume pointer size = 4



Pointer Variables

- Pointer variables have types
  - indicate which type it points to
  - that's why they are named pointers

```
int i = 5;       // define variable i of type int, its value is an integer
int *ip; = &i;   // define variable ip of type int*
                 // its value is a memory address where an int resides at
                 // or, we say ip points to int, or ip is a pointer to int
double d = 3.0;
double *dp = &d;
dp = &i;         // error! dp is a pointer to double, NOT to int
                 // Why? C++ is a language with very strong type-checking
                 // there is one more reason, discuss later
int **ipp = &ip;  //ok, ipp is a pointer to a pointer to int
```

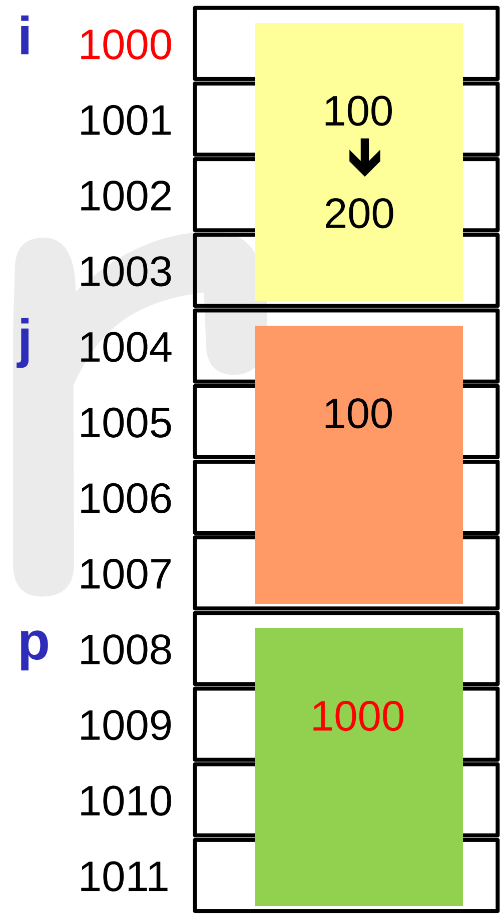

Unary Operators & and *

- Unary address-of (or reference) operator &
  - get the address of an object

```
int i = 100;   // address of i is 1000
               // value stored in i is 100
int p = &i;    // get address of i , then store in p
               // address of p is 1008
               // value stored in p is 1000 (i's address)
```

- Unary dereference operator *
  - refer to the object a pointer points to

```
int j = *p;    // *p refer to i
               // ➔ equivalently, int j = i;
*p = 200;      // similarly, ➔ i = 200;
```



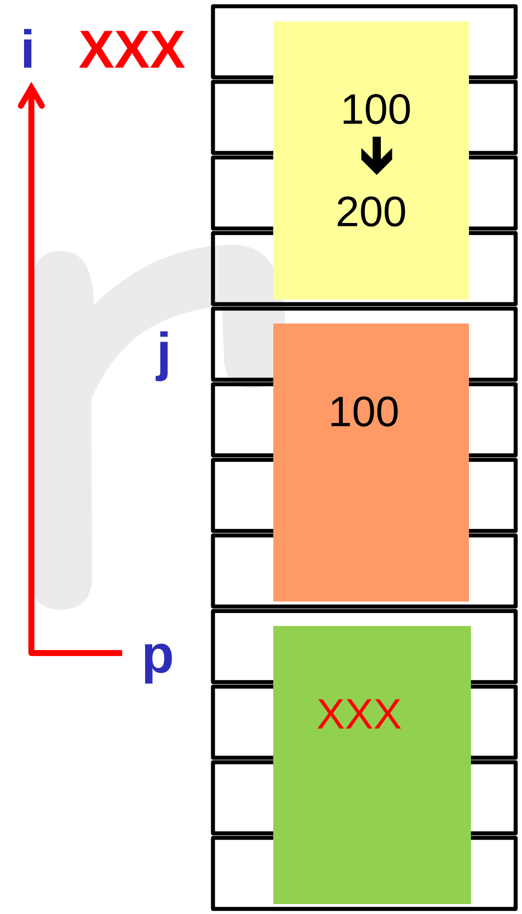
Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Abstraction of Pointer Value

- Unary address-of (or reference) operator &
  - get the address of an object

int i = 100; // address of i is XXX
// **I don’t care what XXX is**
int p = &i; // get address of i , then store in p
// value stored in p is XXX
// **again, I don’t care what XXX is**
// What’s important? ➔ **p points to i**



- Unary dereference operator *
  - refer to the object a pointer points to

int j = *p; // p points to i ➔ *p refer to i
// ➔ equivalently, int j = i;
*p = 200; // similarly, ➔ i = 200;
// **no need to know what the value of p is !**

Pointer vs. int

- Pointer value is an address
- Address value is an integer
- However, pointer value is **NOT** an integer!
  - for abstraction and preventing careless coding errors!

```
int i = 100, *pi = &i, j;
j = pi;  // Oops, I actually mean *pi. But don't worry! error here!
```

- In C/C++, pointer and int are **NOT** interchangeable intentionally

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Put Them Together

```
void f(int* x, int y, int& z) {
    *x = y;         // * : dereference
    y = z;
    z = *x;         // * : dereference
}

void g() {
    int i = 10, j = 20;
    i = i * 10;     // * : multiply
    int k = i & j;  // & : bitwise and ; k is 4
    int *pi;        // define pi as a pointer to int
    int& rj = j;    // define rj as a reference of j (of type int)
    pi = &i;        // & : address-of
    f(pi, rj, k);   // i = 20, j = 20, k = 20
}
```

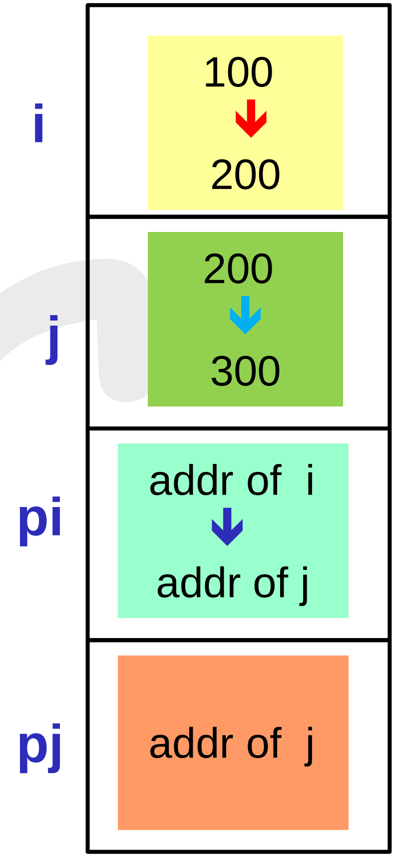

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Pointer Assignment

```
int i = 100, j = 200, *pi = &i, *pj = &j;
int m = 10, n = 20;

m = n;          // replace the value in m with
                // the value in n ➔ m = 20
*pi = *pj;      // ➔ i = j ➔ replace the value in i
                // with the value in j ➔ i = 200
pi = pj;        // replace the value in pi with
                // the value in pj
                // ➔ pi points to what pj points to
*pi = 300;      // ➔ j = 300
```

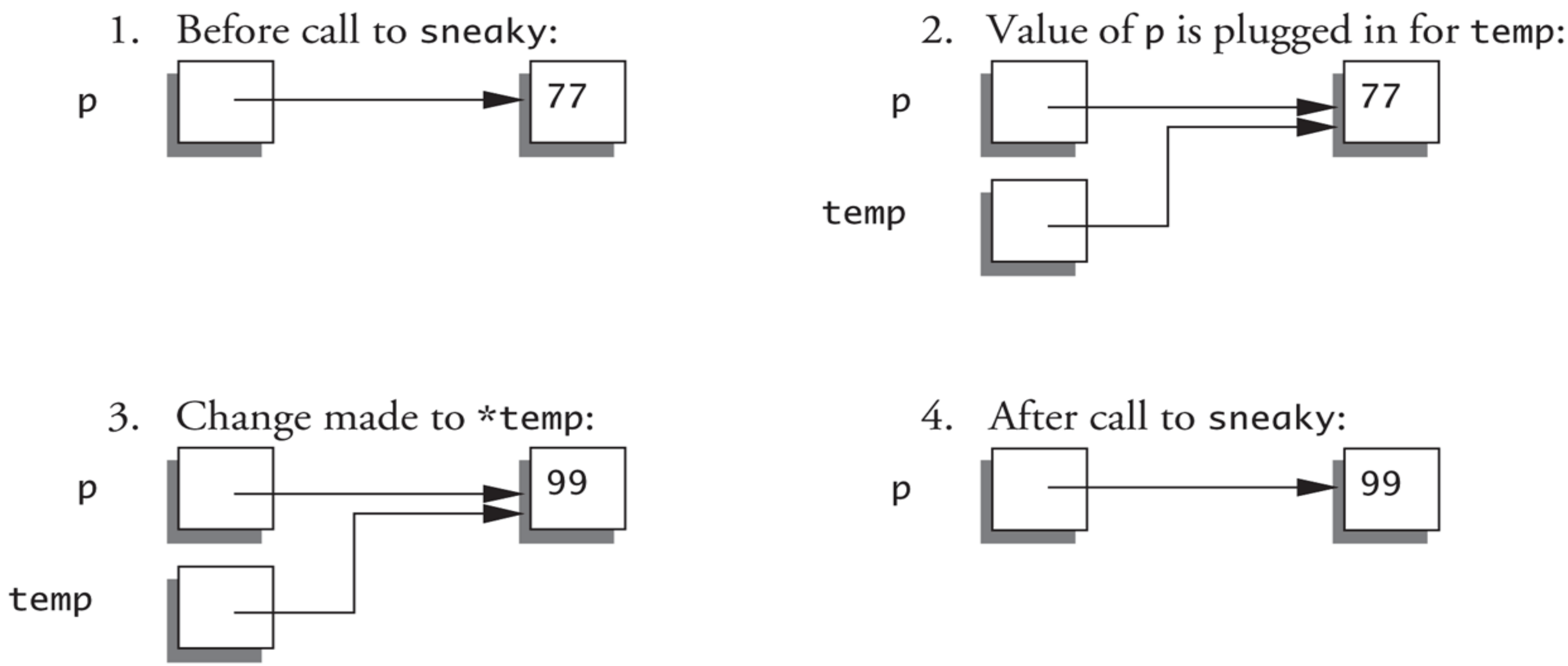


Behind the Theme: Call-by-Pointer-Value

```
void sneaky(int *temp) { // the value of temp is 1000 ; just call-by-value
    cout << *temp << endl;          // 77
    *temp = 99;
}

void f() {
    int i = 77;      // assume the address of i = 1000
    int *p = &i;     // the value of p is 1000
    cout << *p << endl;             // 77
    sneaky(p);       // call-by-value
    cout << *p << endl;             // 99
    cout << i << endl;              // 99
}
```

Example: Call-by-Pointer-Value



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Constant Pointer vs. Pointer to Constant (1/2)

```
const int ci = 0, cj = 10;
int i = 20, j = 30;

int * pi = &i;      // pi is a pointer to int i
j = *pi;            // ok
*pi = 40;           // ok, *pi is an int
pi = &j;            // ok, pi now points to int j; pi's value can be changed
pi = &ci;           // error, pi cannot point to constant int

const int *pci = &ci;     // pci is a pointer to constant int ci
j = *pci;                 // ok
*pci = 50;                // error, *pci is a constant int
pci = &cj;                // ok, pci now points to constnat int cj;
                          // pci's value can be changed
pci = &j;                 // still ok, pci can points to int
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Constant Pointer vs. Pointer to Constant (2/2)

```
const int ci = 0, cj = 10;
int i = 20, j = 30;

int * const cpi = &i;         // cpi is a constant pointer to int
int * const cpj = &cj;        // error, cpj cannot point to constant int
j = *cpi;                     // ok
*cpi = 40;                    // ok, *cpi is an int
cpi = &j;                     // error, pci is a constant pointer
pi = &ci;                     // error, pci is a constant pointer

const int * const cpci = &ci;    // cpci is a constant pointer to constant int
const int * const cpcj = &j;     // still ok, cpci can point to int
j = *cpci;                    // ok
*cpci = 50;                   // error, *cpci is a constant int
cpci = &cj;                   // error, cpci is a constant pointer
cpci = &j;                    // error, cpci is a constant pointer
```

Array Name vs. Pointer

- **Array name is actually a constant pointer**

```
void func1(int arr[]);
void func2(int *arr);
void f() {
    int arr1[10], arr2[10];
    int *pi = arr1;          // ok, arr1 is actually a constant pointer to int
    pi[3] = 100;             // ok, arr1[3] = 100, pi acts like an array name
    pi = arr2;               // ok, pi is NOT a constant pointer
    pi[3] = 200;             // ok, arr2[3] = 200, pi acts like an array name
    func1(arr1);             // ok
    func2(arr1);             // ok
    func2(pi);               // ok
    func1(pi);               // ok
    arr1 = arr2;            // error, arr1 is a constant pointer, think about why?
}
```

Pointer Arithmetic (1/3)

- Certain arithmetic operations on pointers are allowed
  - address calculation related arithmetic

```
double da[100], *pd1 = &da[0], *pd2 = &da[1];
int ia[100], *pi1 = &ia[0], *pi2 = &ia[4];
cout << pd1 << '\t' << pd2 << endl;
cout << pi1 << '\t' << pi2 << endl;

int offset1 = pd2 – pd1;
int offset2 = pi2 – pi1;
int offset3 = pi1 – pi2;
cout << offset1 << '\t' << offset2 << '\t' << offset3 << endl;
```

Output: (in my PC)
0x28fc20 0x28fc28
0x28fa80 0x28fa90
? ? ?

Now, you should understand why the type a pointer points to does matter!

Pointer Arithmetic (2/3)

```
int arr[100], *pi = arr; *pj = &arr[50];
```

- Pointer ± integral value (offset)

```
*(pi + 5) = 10;      // arr[5] = 10
*(6 + pi) = 20;      // arr[6] = 20 (not preferred)
pi[7] = 30;          // pi[7] ←→ *(pi + 7) ←→ *(7 + pi) ; arr[7] = 30
*(arr + 8) = 40;     // arr[8] = 40
*++pi = *pj++;       // arr[1] = arr[50] and then pj points to arr[51]
pi += 20;            // pi points to arr[21]

*(pi – 1) = 60;      // arr[20] = 60
*--pi = *pj--;       // arr[20] = arr[51] and then pj points to arr[50]
pi -=20;             // pi points to arr[0]
*(1 – pi) = 70;      // error, meaningless!
*(pi * 3) = 80;      // error, meaningless!
*(pi / 4) = 90;      // error, meaningless!
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Pointer Arithmetic (3/3)

```
int arr1[100], arr2[100], *pi = &arr1[10], *pj = &arr1[50];
double arr3[100];
```

- Pointer – Pointer

```
int offset =  pj – pi;          // 50 – 10 = 40
offset = pi – pj;               // 10 – 50 = -40
offset = pi – arr2;             // ok, but meaningless! Avoid doing so
offset = pi – arr3;             // error, different kinds of pointers
int sum = pi + pj;              // error, meaningless! same for *, /, %, …
```


Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Dynamic Memory in C (1/2)

- Sometimes, there is something we just can't possibly know in advance …

```
void f() {
    int score[100]; // the size is FIXED before program execution
    int num;
    cout << "How many students in this class? : ";
    cin >> num;   // what if num > 100??
    for(int i = 0; i < num; ++i)
        cin >> score[i];   // out-of-range runtime error!
    // …
}
```

Dynamic Memory in C (2/2)

- Well, we don't make decision until we know …

```
void f() {
    int *score; // score is just a pointer
    int num;
    cout << “How many students in this class? : ”;
    cin >> num;
    score = (int*) malloc( sizeof(int) * num );
    // allocate a memory block from system to store num integers
    for(int i = 0; i < num; ++i)
        cin >> score[i];   // score acts like an array! Discuss later
    // …                   // no out-of-range error!
    free(score);  // deallocate (return) the memory block to system
}
// Allocate dynamic memory ➔ void *malloc(size_t size); // size in byte
// Deallocate dynamic memory ➔ void free(void * ptr);
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Preview: Dynamic Memory in C++

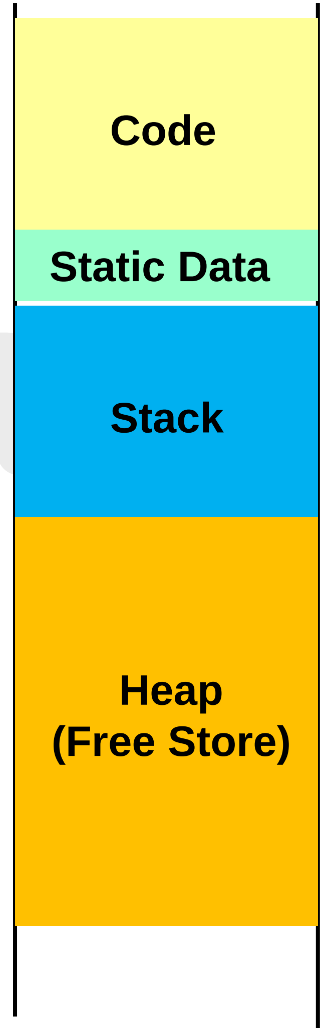
```
void f() {
    int *score; // score is just a pointer
    int num;
    cout << "How many students in this class? : ";
    cin >> num;
    score = new int[num];
    // allocate a memory block from system to store num integers
    // i.e., determine the array size at runtime
    for(int i = 0; i < num; ++i)
        cin >> score[i];   // score acts like an array! Discuss later
    // …
    delete[ ] score;          // deallocate (return) the memory block to system
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Memory Layout

Typical memory layout for a program at runtime

- Code segment
  - program execution code
- Static data segment
  - global objects and static local objects
- **Stack**
  - non-static local objects (i.e., automatic objects)
- **Heap or free sore**
  - **free** memory, not used by program at the beginning
  - its size is **finite** and managed by operating system
- **Dynamic memory management**
  - ask for extra memory blocks from heap by **new** operators **dynamically** (i.e., at runtime)
  - return them back to heap by **delete** operators dynamically



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

new Operators

- Allocate dynamic memory from heap
  - when extra memory is required at runtime
  - get memory from heap
  - if requesting an object of type **X**
    ➔ return value of new operator is of type **X***

```
int *p = new int(5);          // *p = 5 initially
… // do something …
delete p;
```

  - p points to allocated memory

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Failures in new Operations

- In C,

```
int *pi;
if( (pi = (int*) malloc(100 * sizeof(int))) == NULL) { // allocation fails
    /* error handling here */
}
/* proceed normally */
```

- In C++

```
int *pi = new int[100];
```

  – Q1: Are there any chances that dynamic memory allocation fails?
  – A1: Of course!
  – Q2:How to do error handling?
  – A2: Let's discuss this issue in Chap 18: **Exception Handling**

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

delete Operators

- Deallocate dynamic memory
  - when allocated memory is no longer needed
  - return previously allocated memory to heap
- Example

```
int *p;
p = new int(5);      // *p = 5 initially
… // do something …
delete p;            // NO return value for delete operator
```

  - deallocate allocated memory pointed to by p

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Coding Practices

```
void f() {
    int *p1, *p2, *p3;
    p1 = new int;                // allocate one int from heap
    p2 = p3 = new int;           // allocate one int from heap

    *p2 = 100;
    cout << *p3 << endl;         // 100
    p2 = p1;
    *p2 = 200;
    cout << *p1 << endl;         // 200

    delete p1;     // deallocate (return) blue int
    *p1 = 300;     // disaster! do NOT use blue int, it has been deallocated
    p3 = p2;       // there is NO WAY to deallocate green int ! BAD!
                   // so called memory leak (有借要有還)
    delete p3;     // disaster! blue int has been deallocated already
}                  // 不要借五毛，卻還一塊
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Dangling Pointers

- delete p;
  - destroy dynamic memory block pointed by p
  - but p still points there!
    - called dangling pointer
  - if p is still used for dereferencing (*p) after deleting
    - unpredictable results
    - often disaster

- **Never use a dynamic block after deallocated!**

Various Variable Types (1/2)

- Local variables
  - defined in function
  - automatically created (born) when function is called
  - automatically destroyed (died) when function is completed
  - a.k.a. automatic variables, auto variables
  - allocated in stack
- Static local variables
  - defined in function
  - initialized at the first visit
  - destroyed when program terminates
  - allocated in (static) data segment

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Various Variable Types (2/2)

- Global variables
  - defined outside all functions
    (including static data members of classes)
  - created when program starts
    - ctors of global class objects are invoked **BEFORE** main() !
  - destroyed when program terminates
  - allocated in (static) data segment
- Dynamic variables
  - dynamically created with new operators
  - dynamically destroyed with delete operators
  - created and destroyed based on program flow
  - allocated in heap

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

typedef (1/2)

- typedef declares a new name for a type

```
typedef int* Pint;        // Pint is a synonym for int*
Pint p1, p2;              // equivalently → int *p1, *p2;
```

- typedef does **NOT** create a new type
  - Pint is just an alias for int*, **NOT** a new type!
  - that is, both Pint and int* indicate the same type
- Why typedef?
  - convenience
  - portability
  - readability and better documentation

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

typedef (2/2)

- Convenience

```
typedef unsigned long int ULI;
ULI a, b, c ;
ULI func(ULI*, const ULI*);
```

- Portability

```
typedef int int32;      // in a machine, sizeof(int) = 4
typedef long int32;     // in a machine, sizeof(int) = 2, sizeof(long) = 4
int32 a, b, c;          // use int32 in whole program, better portability
```

- Readability and better documentation **(advanced)**

```
typedef  char* (*PFI)(char*, const char*);       // What the hell is this?
PFI  strcat, strcpy;
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Back to Classes: Operator ->

- Member access operator, ->
  - member selection from a pointer

```
class X {
public:
    int d1, d2;
    void mbr_func(int);
// other stuffs
};

void f() {
    X x, *px = &x;
    a.d1 = 10;              // member selection using .
    a.mbr_func(3);
    px->d2 = 20;            // equivalent to ➔ (*px).d2 = 20;
    px->mbr_func(5);        // equivalent to ➔ (*px).mbr_func(5);
}
```

Just for convenience

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

this Pointers

- Member functions may need to refer to calling object

- Use predefined **this** pointer
  - C++ guarantees that **this** pointer points to the calling object in nonstatic member functions
  - why not static member functions?

  - type of **this** pointer of class X:

    in **non-constant** nonstatic member functions ➔ **X* const this**

    in **constant** nonstatic member functions ➔ **const X* const this**

Example: Using this Pointer

- Assignment operator =
  - by C++ grammar, following statements are legal

```
int x;
++(x = 3);    // x = 4 in the end
```

- **this** pointer helps mimic that behavior in user-defined types

```
class X {
    int a, b;
 public:
    X& operator=(const X& rhs) {
        a = rhs.a;
        this->b = rhs.b;          // ok, equivalent to b = rhs.b;
        this += 2;                // error, this is a constant pointer!
        return *this;             // referring to calling object
    }
};
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

More on new Operators (1/2)

X* p = new X;

- If initialization of an object of type X is not mandatory
  - e.g., int, char, float, double, …
  - initialization is optional

```
int *p1 = new int;        // ok, value of *p1 is unknown initially
int *p2 = new int(5);     // still ok; value of *p2 is 5 initially
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

More on new Operators (2/2)

- If initialization of an object of type X is mandatory
  - user-defined types ➔ one of ctors **MUST** be invoked

```
class complex {
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) { }
};
void f() {
    complex *pc = new complex;    // error! object cannot be initialized
    complex *pc = new complex(1.0, 2.0) ;       // ok!
}
```

  - if class X has a default ctor

```
    X *px = new X;     // ok! default ctor is invoked
```

Destructors (1/3)

- At the end of Chap 8, discussions about operator[]

```
class IntArr { // int array with runtime range checking
    int size, *arr;
public:
    IntArr(int sz) :size(sz)  { arr = new int[sz] ;  }     // ctor
    int& operator[](int idx);  // access idx-th element with range checking
    // other stuffs;
};

void f(int val) {
    IntArr ia(100);
    ia[10] = 15;
    ia[20] = ia[10];
    ia[val] = 30;   // runtime error if val < 0 or val >= 100
}
```

Umm, every thing looks perfect …
What?!
There is a memory leak issue here?!
How comes???

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Destructors (2/3)

- ia is a local (auto) variable
  - memory for ia (used by ia.size & ia.arr) is automatically allocated/deallocated when ia is created/destroyed
- However, memory block dynamically allocated in ctor never gets deallocated ➔ memory leak!
- But **ia** is destroyed automatically (and implicitly) as soon as **f** completes…
- When, Where, and How to deallocate that memory???

- Solution ➔ **destructor (dtor)** !

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Destructors (3/3)

```
class IntArr {
    int size, *arr;
public:
    IntArr(int sz) :size(sz)  { arr = new int[sz] ; }    // ctor
    ~IntArr();                                             // dtor
    int& operator[](int idx);  // access idx-th element with range checking
    // other stuffs;
};

IntArr::~IntArr(      ) { // no return type and no parameters are allowed
    detele[] arr;   // deallocation here, no memory leak now
}
```

- dtor is automatically invoked right before an object is destroyed
  – mainly for **clean-up** operations

ctor vs. dtor

- ctor is automatically invoked when an object is created
- dtor is automatically invoked when an object is destroyed

- both have no return type
- ctor can take parameters while dtor cannot

- A class can have as many ctors as it wants
- A class can only have one dtor

- Both cannot be static member functions
- Both cannot be constant member functions

What Compiler silently Do for You (1/2)

```
class Empty {
    int i;
    ABC a;          // ABC is a class with default ctor and dtor
    XYZ x;          // XYZ is a class with default ctor and dtor
};
```

- No member functions for class Empty

```
void f() {
    Empty e, f;     // default ctor is required
    Empty g(e);     // copy ctor is required
    f = e;          // copy assignment operator is required
}
```

- To your surprise, in most cases, above code can compile!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

What Compiler silently Do for You (2/2)

- If a class has no ctors at all, compiler will try to generate a public default ctor
  - its behavior is to invoke default ctor for every class data member
- If a class has no copy ctor, compiler will try to generate a public one
  - its behavior is to do member-wise copy
- If a class has no copy assignment operator, compiler will try to generate a public one
  - its behavior is to do member-wise copy assignment
- if a class has no dtor, compiler will try to generate a public one
  - its behavior is to invoke dtor for every class data member

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example Case

Sometimes, compiler generates what we wants

```
class A {
public:
    A() { cout << "ctor of A is called.\n"; }
    ~A() { cout << "dtor of A is called.\n"; }
};

class B {
public:
    B() { cout << "ctor of B is called.\n"; }
    ~B() { cout << "dtor of B is called.\n"; }
};

class C {  int i; A a; B b; int j;  }; // compiler generates public default ctor and dtor for C

int main()
{
    C c;
    return 0;
}
```

Output:
=========
ctor of A is called.
ctor of B is called.
dtor of B is called.
dtor of A is called.

ctors are called in declaration order
dtors are called in reverse order

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Pointers & Dynamic Memory

Copy ctor and Assignment Operator (1/3)

However, sometimes, what compiler implicitly generates for us are simply disasters …

```
class IntArr {
    int size, *arr;
public:
    IntArr(int sz) :size(sz)  { arr = new int[sz] ;  }     // ctor
    ~IntArr() {  delete[] arr;  }                          // dtor
    int& operator[](int idx);
    // other stuffs, but no copy ctor, no copy assignment operator
};  // compiler generates public copy ctor and copy assignment operator

void func() {
    IntArr a(100), b(100);
    // set values for elements of a
    b = a;  // use copy assignment operator generated by compiler, disaster!
    a[20] = 100;  // also, b[20] = 100,  disaster!
    IntArr c(a);  // use copy ctor generated by compiler, disaster!
} // 2 big and 2 not-so-big disasters right before func() completes
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Copy ctor and Assignment Operator (2/3)

- Compiler-generated copy ctor and copy assignment operator don’t work as expected if dynamic memory is in use
- In that case, define correct ones yourself!

```
class IntArr {
    int size, *arr;
public:
    IntArr(int sz) :size(sz)  { arr = new int[sz] ; }     // ctor
    ~IntArr() {  delete[] arr;  }                          // dtor
    IntArr(const IntArr&);                                 // copy ctor
    IntArr& operator=(const IntArr&);                      // copy assignment operator

    int& operator[](int idx);
    // other stuffs
};
```


Pointers & Dynamic Memory

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Copy ctor and Assignment Operator (3/3)

```
IntArr::IntArr(const IntArr& src)    // copy ctor
: size(src.size)  {
    arr = new int[size];
    for (int i = 0; i < size; ++i)
        arr[i] = src.arr[i];
}

IntArr& IntArr::operator=(const IntArr& rhs)  {  // copy assignment operator
    size = rhs.size;
    int *pi = arr;
    arr = new int[size];
    for (int i = 0; i < size; ++i)
            arr[i] = rhs.arr[i];
    delete[ ] pi;
    return *this;
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Self Assignment Issue

- Another version for copy assignment operator

```
IntArr& IntArr::operator=(const IntArr& rhs)  {  // copy assignment operator
    size = rhs.size;
     delete[ ] arr;
    arr = new int[size];
    for (int i = 0; i < size; ++i)
            arr[i] = rhs.arr[i];
    return *this;
}
```

- Purple version seems good and a bit faster than green version …
- However, what if …

```
void func() {
    IntArr a(100);
    a = a;  // self assignment, it's silly but legal
}
```

Beware of self assignment!

Green version is actually superior to purple one

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Arrays of Class Objects

- Array
  - e.g., int arr[20];
  - size is fixed at compile time ➔ inflexible
  - however, allocation is very time-efficient
- Array of class objects
  - every element in array must be properly constructed
  - i.e., default ctor is required to define an array of class objects
  - when created, default ctor is called for every element in order
  - when destroyed, dtor is called for every element in reverse order

```
void func() {    // assume class X has a default ctor and class Y doesn't
    X a[20];   // ok, call default ctor of X for a[0], a[1], …, a[19]
    Y b[20];   // error, Y has no default ctor
    // do something
} // when func completes, call dtor of X for a[19], a[18], …, a[0]
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Dynamic Arrays

- **Dynamic** array
  - e.g., $\boxed{\text{int *pi = new int[size];}}$
  - size can be determined at runtime ➔ flexible
  - however, dynamic memory allocation is relatively time-consuming

Avoid using dynamic array if size is known at compile time

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Creating Dynamic Arrays

- Use new[] operator ➔ X *pX = new X[sz];
  - sz can be a variable, determined at runtime
  - dynamically allocate enough memory for sz elements of X
  - return the start address with type X*
  - if X is a user-defined type, **default ctor of X** is invoked for every element in this dynamic array in order
  - what if there is no default ctor? ➔ new[] is not allowed!

```
int sz = 10;
double *pd = new double[sz]; // ok, double is a built-in data type

// assume class X has default ctor and class Y doesn't
X *pX = new X[sz];    // ok, call default ctor for pX[0], pX[1], …, pX[sz-1]
Y* pY = new Y[sz];    // error! no default ctor is available
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Destroying Dynamic Arrays

- Use delete[] operator ➔ delete[] pX;
  - no need to put sz in []
  - need [] to tell compiler that pX points to an array instead of a single object
  - dynamically deallocate previously-allocated memory
  - no return value
  - if X is a user-defined type, **dtor of X** is invoked for every element in this dynamic array in reverse order

```
delete[] pd;

delete[] pX;   // call dtor for pX[sz-1], pX[sz-2], … pX[0]
```

Multidimensional Dynamic Arrays (1/2)

Advanced

- Yes, we can!
- Two ways to make an mxn 2D dynamic array
  - **1st:** an array of m pointers, each one points to an array of n elements
  - **2nd:** an array of m*n elements

- First method

How about 3D dynamic array?

```
void f(int m, int n) {
    int**ppi = new int*[m];
    for(int i = 0; i < m; ++i)
        ppi[i] = new int[n];  // ppi[i] is a pointer to an array of n ints
    //  ....   ,   here, treat ppi as int ppi[m][n]
    for(int i = 0; i < m; ++i) // deallocation
        delete[] ppi[i];
    delete[] ppi;              // (m+1) new[]/delete[] pairs in total
}
```


Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Multidimensional Dynamic Arrays (2/2)

Advanced

- Second method

How about 3D dynamic array?

```
class Int2D {
    int m, n, *begin;
public:
    Int2D(int x, int y) : m(x), n(y) { begin = new int[m*n];  }
    ~Int2D() { delete[] begin; }
    int& operator()(int idx1, idx2) { return *(begin + idx1 * n + idx2) ;  }
    // other stuffs, only one new[]/delete[] pair
};
void f(int m, int n) {
    Int2D  arr(m, n);
     // here, treat arr as int arr[m][n]
    arr(2, 3) = 100; arr(4, 5) = arr(2,3);
    ++arr(4,5); // ...
}
```

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Pointers & Dynamic Memory

Overload new and delete

Advanced

- Yes, of course, new, new [ ], delete, and delete [ ] can all be overloaded within a class
  - they are overloaded mainly for better runtime efficiency

- However, they are advanced topics and beyond the scope of this course

Summary (1/2)

- Pointer vs. Address
- Pointer operations (address-of, dereference, assignment)
- Constant pointer vs. Pointer to constant
- Array name is constant pointer
- Pointer arithmetic and pointer offset
- Dynamic memory and heap
- Operators: new, new[], delete, detele[]
- Local, static local, global, dynamically-allocated variables

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Summary (2/2)

- Operator -> ; **this** pointer
- new ➔ ctor ; delete ➔ dtor
  - for proper object construction and destruction
  - especially when dynamic memory is in use
- Revisit ctor, copy ctor, copy assignment operator, and dtor
  - for proper dynamic memory management
- (Dynamic) array of class objects
  - ctor in order and dtor in reverse order
- Multi-dimensional dynamic arrays
  - 2 alternatives

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Summary of Guidelines

- Use new/delete instead of malloc/free
- Use the same form in corresponding uses of new & delete
  - new ←→ delete ; new[] ←→ delete[]
- Beware of memory leak
- Never use a dynamic block after deallocated!
- Avoid using dynamic array if size is known at compile time
- Do necessary delete/delete[] on pointer members in dtors
- Define copy ctor, copy assignment operator, and dtor for classes with dynamically allocated memory
- Handle self assignment when overloading operator=

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw