Introduction to Object-Oriented Programming

COMP2011: C++ Pointers & Dynamic Data

Dr. Cindy Li
Dr. Brian Mak
Dr. Dit-Yan Yeung

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

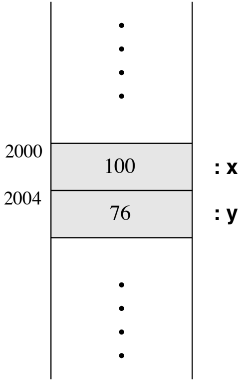
Part I

lvalue (Address) and rvalue (Content)

Variables

A variable is a symbolic name assigned to some memory storage.

- The size of this storage depends on the type of the variable. e.g. char is 1-byte long and int is 4-byte long.
- The difference between a variable and a literal constant is that a variable is addressable.
- e.g. $\boxed{x = 100;}$ x is a variable and 100 is a literal constant.



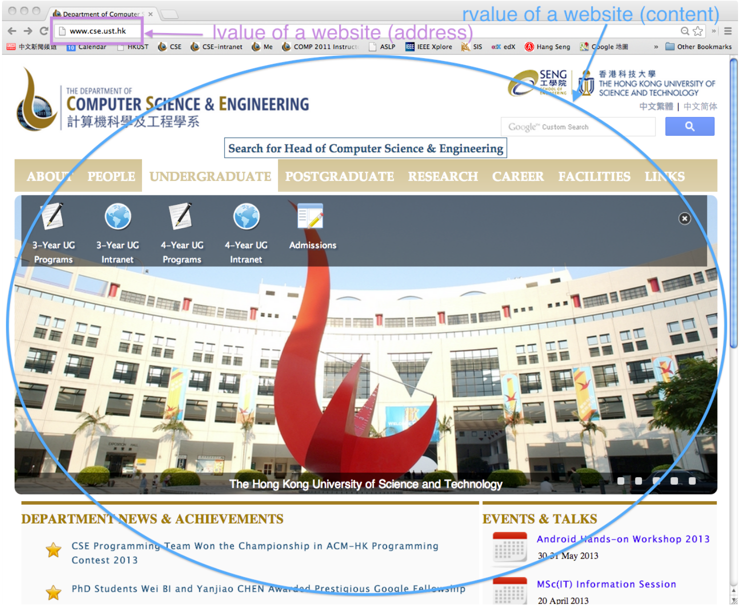
Example: lvalue and rvalue

$x = x + 1;$

- A variable has dual roles. Depending on where it appears in the program, it can represent an
  - lvalue: location of the memory storage (read-write)
  - rvalue: value in the storage (read-only)
- They are so called because a variable represents an lvalue (rvalue) if it is written to the left (right) of an assignment statement.
- Which of the following C++ statements are valid? Why?

```
int x;
4 = 1;
(x + 10 ) = 6;
cout << ++++++x << endl;        // ANSI C++ Ref. Section 5.3.2
cout << x++++++ << endl;        // ANSI C++ Ref. Section 5.2.6
```

Analogy: Website Has an lvalue and a rvalue Too!



Get the Address by the Reference Operator &

Syntax: Get the Address of a Variable

& <variable>

```
#include <iostream>                                /* File: var-addr.cpp */
using namespace std;

int main(void)
{
    int x = 10, y = 20;
    short a = 9, b = 99;

    cout << "x = " << x << '\t' << "address of x = " << &x << endl;
    cout << "y = " << y << '\t' << "address of y = " << &y << endl;
    cout << "a = " << a << '\t' << "address of a = " << &a << endl;
    cout << "b = " << b << '\t' << "address of b = " << &b << endl;
    return 0;
}
```

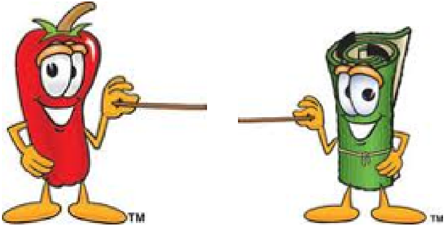
Example: Address of Formal Parameters

```
#include <iostream>                              /* File: fcn-var-addr.cpp */
using namespace std;

void f(int x2, int& y2)
{
    short a = 9, b = 99;
    cout << endl << "Inside f(int, int&)" << endl;
    cout << "x2 = " << x2 << '\t' << "address of x2 = " << &x2 << endl;
    cout << "y2 = " << y2 << '\t' << "address of y2 = " << &y2 << endl;
    cout << "a = " << a << '\t' << "address of a = " << &a << endl;
    cout << "b = " << b << '\t' << "address of b = " << &b << endl;
}

int main(void)
{
    int x = 10, y = 20;
    cout << endl << "Inside main(void)" << endl;
    cout << "x = " << x << '\t' << "address of x = " << &x << endl;
    cout << "y = " << y << '\t' << "address of y = " << &y << endl;
    f(x, y);
    return 0;
}
```

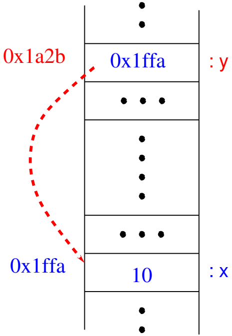
Question: Can you see the difference between PBV and PBR?

Part II

What is a Pointer?



Pointer Variable



Syntax: Pointer Variable Definition

<type>* <variable>;

- A pointer variable stores the address of another variable.
- If variable *y* stores the address of variable *x*, we say "*y* points to *x*."
- Notice that a pointer variable is just a variable which has its *own* address in memory.

```cpp
#include <iostream>                                   /* File: pointer-var.cpp */
using namespace std;
int main(void)
{
    int x = 10; int* y = &x;              // y now contains the address of x
    cout << "x = " << x << '\t' << "address of x = " << &x << endl;
    cout << "y = " << y << '\t' << "address of y = " << &y << endl;
    return 0;
}
```

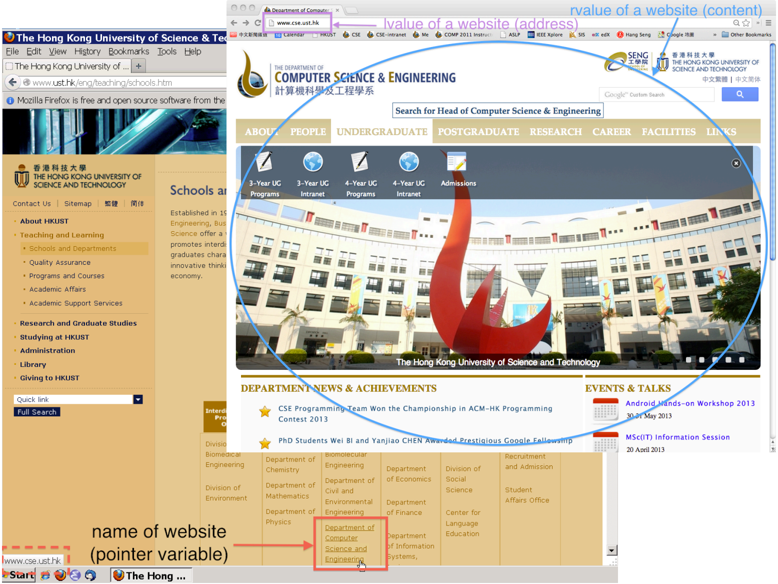
Get the Content by the Dereference Operator *

Syntax: Get the Content Through a Pointer Variable

*<pointer variable>

```
#include <iostream>                                   /* File: pointer-deref.cpp */
using namespace std;

int main(void)
{
    int x = 10, z = 20;
    int* y = &x;                            // y now contains the address of x
    cout << "x = " << x << '\t' << "address of x = " << &x << endl;
    cout << "z = " << z << '\t' << "address of z = " << &z << endl;
    cout << "y = " << y << '\t' << "address of y = " << &y << endl;

    z = *y;   // Get the content from the address stored in y, put it to z
    cout << endl;
    cout << "z = " << z << '\t' << "address of z = " << &z << endl;
    cout << "y = " << y << '\t' << "*y = " << *y << endl;
    return 0;
}
```

Analogy: Name of a Website as a Pointer Variable



Web Analogy

WEB	C++
anchor text	pointer variable
web address (URL)	lvalue of a variable (variable address)
web content	rvalue of a variable (variable's value)
click on the anchor text	dereference a pointer variable

Example: Pointer Manipulation

```cpp
#include <iostream>                                    /* File: pointer.cpp */
using namespace std;

int main(void)
{
    int x1 = 10, x2 = 20;
    int *p1 = &x1;                                     // p1 now points to x1
    int *p2 = &x2;                                     // p2 now points to x2

    *p1 = 5;                                           // now x1 = 5
    *p2 += 1000;                                       // now x2 = 1020
    *p1 = *p2;      // now *p1 = *p2 = x1 = x2 = 1020, but p1 != p2
    p1 = p2;                          // now p1 and p2 both point to x2

    cout << "x1 = " << x1 << '\t' << "&x1 = " << &x1 << endl;
    cout << "x2 = " << x2 << '\t' << "&x2 = " << &x2 << endl;
    cout << "p1 = " << p1 << '\t' << "*p1 = " << *p1 << endl;
    cout << "p2 = " << p2 << '\t' << "*p2 = " << *p2 << endl;
    return 0;
}
```

Example: Pointer and sizeof( )

```
#include <iostream>                                    /* File: pointer-sizeof.cpp */
using namespace std;

int main(void)
{
    char c = 'A'; short s = 5; int i = 10; double d = 5.6;
    char* pc = &c; short* ps = &s; int* pi = &i; double* pd = &d;

    cout << sizeof(pc) << '\t' << sizeof(*pc) << '\t' << sizeof(&pc) << endl;
    cout << sizeof(ps) << '\t' << sizeof(*ps) << '\t' << sizeof(&ps) << endl;
    cout << sizeof(pi) << '\t' << sizeof(*pi) << '\t' << sizeof(&pi) << endl;
    cout << sizeof(pd) << '\t' << sizeof(*pd) << '\t' << sizeof(&pd) << endl;

    return 0;
}
```

What can a Pointer Point to?

A pointer can point to

- objects of basic types: char, short, int, long, float, double, etc.
- objects of user-defined types: struct, class (discussed later)
- another pointer!
- even to a function $\Rightarrow$ function pointer!

Example: Pointer to Pointer to Pointer ...

```
#include <iostream>                                  /* File: pointer-pointer.cpp */
using namespace std;

int main(void)
{
    int x = 16;
    int* xp = &x;                                                          // xp -> x
    int** xpp = &xp;                                                // xpp -> xp -> x
    int*** xppp = &xpp;                                     // xppp -> xpp -> xp -> x

    cout << "x address = " << &x << " x = " << x << endl;

    cout << "xp address = " << &xp << " xp = " << xp
         << " *xp = " << *xp << endl;

    cout << "xpp address = " << &xpp << " xpp = " << xpp
         << " *xpp = " << *xpp
         << " **xpp = " << **xpp << endl;

    cout << "xppp address = " << &xppp << " xppp = " << xppp
         << " *xppp = " << *xppp
         << " **xppp = " << **xppp
         << " ***xppp = " << ***xppp << endl;
    return 0;
}
```

Variable, Reference Variable, Pointer Variable

```cpp
#include <iostream>                                     /* File: confusion.cpp */
using namespace std;
int x = 5;                                              // An int variable
int& xref = x;            // A reference variable: xref is an alias of x
int* xptr = &x;                 // A pointer variable: xptr points to x

void xprint(void)
{
    cout << hex << endl;            // Print numbers in hexadecimal format
    cout << "x = " << x << "\t\tx address = " << &x << endl;
    cout << "xref = " << xref << "\t\txref address = " << &xref << endl;
    cout << "xptr = " << xptr << "\txptr address = " << &xptr << endl;
    cout << "*xptr = " << *xptr << endl;
}

int main(void)
{
    x += 1; xprint( );
    xref += 1; xprint( );
    xptr = &xref; xprint( );                    // Now xptr points to xref
    return 0;
}
```

const Pointer

Syntax: const Pointer Definition

<type>* const <pointer variable> = &<another variable>;

- A const pointer must be initialized when it is defined; just like any C++ constant.
- A const pointer, once initialized, cannot be changed to point to something else.
- However, you are free to change the content in the address it points to.

Example: const Pointer

```
int x = 10, y = 20;
int* const xcp = &x;
xcp = &y;        // Compile Error: a const pointer!
*xcp = 5;        // Compile Okay: what it points to is not const
```

Pointer to const Objects

Syntax: Definition of Pointer to a const Object

const <type>* <pointer variable>;

- It is not necessary to initialize a pointer to const object when it is defined, though you may.
- You are free to change the pointer itself to point to different objects during program execution.
- However, the content of the object pointed to by such pointer cannot be changed through the pointer. But the content of the object can still be changed by the object directly!
- Analogy: In a sense, the anchor texts on a webpage that allow you to surf other websites are pointers to const webpages since, in your perspective, you cannot change the content of those webpages.

Example: Pointer to const Object

```
int x = 10, y = 20;
const int* pc = &x;
pc = &y;        // Compile Okay: pc is free to point to x, y, z, or any int
*pc = 5;        // Compile Error: its content is const when accessed thru pc!
y = 8;          // Compile Okay: y is not a const object
```

PBR = PBV + Pointer

- The programming language C only have one way to pass arguments to a function, which is PBV.
- To simulate the effect of PBR, one may pass the address of an object to a function.
- Inside the function, the object is represented by a pointer.
- Then one may change the object's value by dereferencing the object's pointer inside the function.

Example: Swap using PBV + Pointer

```
#include <iostream>                                    /* File: pbv-pointer.cpp */
using namespace std;

void swap(int* x, int* y)
{
    cout << "x = " << x << "\t*x = " << *x << endl;
    cout << "y = " << y << "\t*y = " << *y << endl << endl;

    int temp = *x;
    *x = *y;
    *y = temp;

    cout << "x = " << x << "\t*x = " << *x << endl;
    cout << "y = " << y << "\t*y = " << *y << endl << endl;
}

int main(void)
{
    int a = 10, b = 20;
    cout << "a = " << a << "\t\t\t&a = " << &a << endl;
    cout << "b = " << b << "\t\t\t&b = " << &b << endl << endl;
    swap(&a, &b);
    cout << "a = " << a << "\t\t\tb = " << b << endl;
    return 0;
}
```

Common Uses of Pointer

- Indirect addressing (c.f. anchor text)
- Dynamic object creation/deletion
- Advanced uses that will not be covered in this course:
  - writing generic functions that can work on any data type (e.g. a sorting function that sorts any data type)
  - implementation of object-oriented technologies such as
    - inheritance,
    - polymorphism (virtual function).

Part III

Pointer to Structure

- You may also define a pointer variable for a struct object.
- Two ways to access struct members through a pointer:
  1. Dereference the pointer and use the . operator.

```
Point a;                 // a constains garbage
Point* ap = &a;          // Now ap points to a

// Dereference ap, then access members by the . operator
(*ap).x = 3.5;
(*ap).y = 9.7;
```

  2. Directly use the → operator.

```
Point a;                 // a constains garbage
Point* ap = &a;          // Now ap points to a

// No dereferencing when using the -> operator
ap->x = 3.5;
ap->y = 9.7;
```

Example: Euclidean Distance Again — point-test.cpp

```
#include <iostream>                                /* File: point-test.cpp */
#include "point.h"
using namespace std;

// A function that computes and prints the Euclidean distance between 2 points
void print_distance(const Point*, const Point*);

int main(void)          /* To find the length of the sides of a triangle */
{
    Point a, b, c;
    cout << "Enter the co-ordinates of point A: "; cin >> a.x >> a.y;
    cout << "Enter the co-ordinates of point B: "; cin >> b.x >> b.y;
    cout << "Enter the co-ordinates of point C: "; cin >> c.x >> c.y;

    print_distance(&a, &b);
    print_distance(&b, &c);
    print_distance(&c, &a);
    return 0;
}
/* g++ -o point-test point-test.cpp point-distance.cpp */
```

```
#include <iostream>                              /* File: point-distance.cpp */
#include <cmath>
#include "point.h"
using namespace std;

double euclidean_distance(const Point* p1, const Point* p2)
{
    double x_diff = p1->x - p2->x;
    double y_diff = p1->y - p2->y;
    return sqrt(x_diff*x_diff + y_diff*y_diff);
}

void print_point(const Point* p)
{
    cout << '(' << p->x << ", " << p->y << ')';
}

void print_distance(const Point* p1, const Point* p2)
{
    cout << "Distance between "; print_point(p1);
    cout << " and "; print_point(p2);
    cout << " is " << euclidean_distance(p1, p2) << endl;
}
```

Example: sort-student-record.cpp Again

```
#include "student-record.h"          /* File: sort-student-record.cpp */
#include "student-record-extern.h"

int main(void)
{
    Student_Record sr[ ] = {
        { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
        { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
        { "Cathy", 10000, 'F', ECE, { 2006 , 8 , 20 } } };

    Date d;                                   // Modify the 3rd record
    set_date(&d, 1980, 12, 25);
    set_student_record(&sr[2], "Jane", 18000, 'F', CSE, &d);

    sort_3SR_by_id(sr);
    for (int j = 0; j < sizeof(sr)/sizeof(Student_Record); j++)
        print_student_record(&sr[j]);
    return 0;
}
/* g++ -o sort-sr sort-student-record.cpp student-record-functions.cpp
   student-record-swap.cpp */
```

Example: student-record-swap.cpp Again

```
#include "student-record.h"          /* File: student-record-swap.cpp */

void swap_SR(Student_Record* x, Student_Record* y)
{
    Student_Record temp = *x;
    *x = *y;
    *y = temp;
}


void sort_3SR_by_id(Student_Record sr[ ])
{
    if (sr[0].id > sr[1].id) swap_SR(&sr[0], &sr[1]);
    if (sr[0].id > sr[2].id) swap_SR(&sr[0], &sr[2]);
    if (sr[1].id > sr[2].id) swap_SR(&sr[1], &sr[2]);
}
```

Example: student-record-functions.cpp Again

```
#include <iostream>                              /* File: student-record-functions.cpp */
#include "student-record.h"
using namespace std;

void print_date(const Date* date)
{
    cout << date->year << '/' << date->month << '/' << date->day << endl;
}

void print_student_record(const Student_Record* x)
{
    cout << endl;
    cout.width(12); cout << "name:  " << x->name << endl;
    cout.width(12); cout << "id:  " << x->id << endl;
    cout.width(12); cout << "gender:  " << x->gender << endl;
    cout.width(12); cout << "dept:  " << dept_name[x->dept] << endl;
    cout.width(12); cout << "entry date:  "; print_date(&x->entry);
}

void set_date(Date* x, unsigned int year, unsigned int month, unsigned int day)
{
    x->year = year; x->month = month; x->day = day;
}

void set_student_record(Student_Record* a, const char name[ ], unsigned int id, char gender, Dept dept, const
Date* date)
{
    strcpy(a->name, name);
    a->id = id;
    a->gender = gender;
    a->dept = dept;
    a->entry = *date;                                   // struct-struct assignment
}
```

Example: student-record-extern.h Again

```
/* File: student-record-extern.h */

void print_date(const Date*);
void print_student_record(const Student_Record*);
void set_date(Date* x, unsigned int, unsigned int, unsigned int);
void set_student_record(Student_Record*, const char[ ], unsigned
int, char, Dept, const Date*);
void swap_SR(Student_Record*, Student_Record*);
void sort_3SR_by_id(Student_Record sr[ ]);
```

Part IV

Dynamic Memory/Objects Allocation and Deallocation

Example: Static Objects

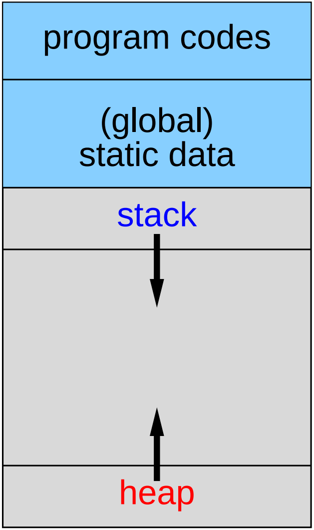
```
float a = 2.3;                          // Global float variable

int main(void)
{
    int x = 5;                          // Local int variable
    char s[16] = "hkust";               // Local char array
    return 0;
}
```

- Up to now, all (local and global) variables you use require static memory allocation: their memory are allocated by the compiler during compilation.
- When these variables — static objects — go out of their scope, their memory are released automatically back to the computer's memory store (RAM).
- Question: What if you want to create an object, or an array whose size is unknown until a user specifies at runtime?

Dynamic Objects

- C++ allows you to create an object, or an array of objects — dynamic objects — on-the-fly at runtime.
- The memory of dynamic objects
  - has to be allocated at runtime explicitly by you,
    ⇒ using the operator new.
  - will persist even after the object goes out of scope.
  - has to be deallocated at runtime explicitly by you,
    ⇒ using the operator delete.
- Static objects are managed using a data structure called stack.
- Dynamic objects are managed using a data structure called heap.

Memory Layout of a C++ Program

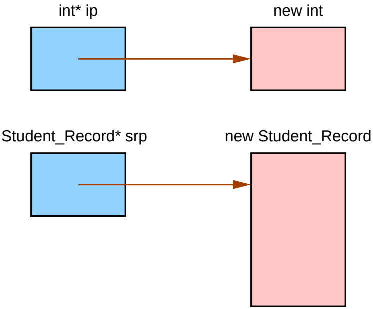


Syntax: Dynamic Memory Allocation Using new

<type>* <pointer-variable> = new <type>;

Examples: Use of the new Operator

```
int* ip = new int;
*ip = 5;
Date d19970701 = { 1997, 7 , 1 };
Student_Record* srp = new Student_Record;
set_student_record(*srp, "Chris", 100, 'M', CSE, d19970701);
```



Dynamic Memory Allocation: Operator new ..

For the line: `int* ip = new int;`

- The computer finds from the heap an amount of memory equal to sizeof(int) and gives it to your program.
- The new operator, which is actually a function, will return a value which is the address of the starting location of that piece of memory.
- That piece of memory is unnamed, and you need to use an int pointer variable (here, ip) to point to it — holding its address (that is returned by the new operator).
- There is *no* other way to access the unnamed memory allocated by the operator new except through the pointers.

For the line: `Student_Record* srp = new Student_Record;`

- The computer gives you an amount of unnamed memory equal to sizeof(Student_Record) from the heap.
- You need to hold its address using a Student_Record pointer variable (here, srp).
- Notice that the variables, ip and srp, are static objects.
- Only the unnamed memories returned by the new operator are dynamic objects.
- Both local static objects and dynamic objects come and go.
- However, the stack will allocate and deallocate local static objects automatically for you.
- But you have to manage the allocation and deallocation of dynamic objects yourselves.

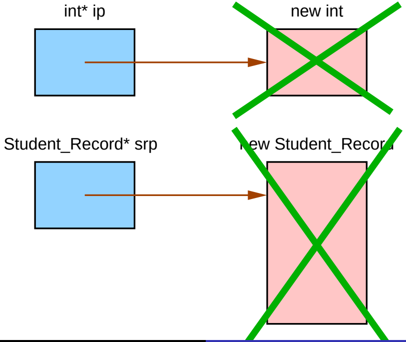
Dynamic Memory Deallocation: Operator delete

Syntax: Dynamic Memory Deallocation Using delete

delete <pointer-variable>;

Examples: Use of the delete Operator

```
delete ip;          // ip is now a dangling pointer
ip = NULL;          // ip is now a NULL pointer
delete srp;         // srp is now a dangling pointer
srp = 0;            // srp is now a NULL pointer
```



Common Bug I: Dangling Pointer — Case 1

- Operator delete releases memory pointed to by a pointer variable (here, ip or srp) back to the heap for recycle.
- However, after the delete operation, the pointer variable still holds the address of the previously allocated unnamed memory.
- Now the pointer becomes a dangling pointer.
- A dangling pointer is a pointer that points to a location whose memory is deallocated.
- Runtime error usually occurs when you try to dereference a dangling pointer either because
  - the memory is no long accessible as it is taken back.
  - the memory has already been recycled and is re-allocated to some other functions or even other programs!

Common Bug I: Dangling Pointer — Case 1 ..

- Modifying the object a dangling pointer points to leads to unpredictable results that usually end up in a program crash.
- To play safe, reset a dangling pointer to a NULL pointer by setting its value to NULL or equivalently 0.
- A NULL pointer is used to indicate a pointer that has not been set to point to something useful.
- Good practices:
  1. Always initialize a pointer to NULL when defining a pointer variable.
  2. Always check whether a pointer is a NULL pointer before using it.

Common Bug I: Dangling Pointer — Case 2

Example: Dangling Pointer

```
/* File: dangling-pointer.cpp */
int* create_and_init(int value) {
    int x = value;                          // x is a local variable
    int* p = &x;                            // p is also a local variable
    return p;
}
int main(void) {
    int* ip = create_and_init(10);
    cout << *ip << endl;
    return 0;
}
```

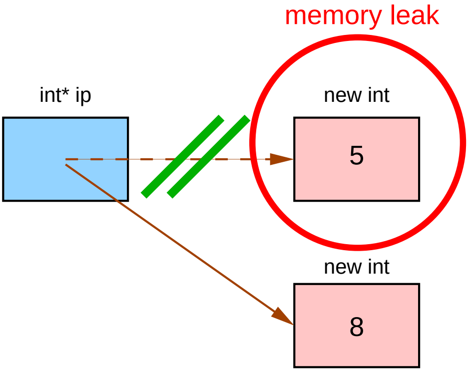
- Local pointer variable, p is pointing to another local variable, x. Both are automatically allocated when the function create_and_init( ) is called, and are automatically deallocated when create_and_init( ) returns.
- Question: What does the pointer variable, ip point to after the call to create_and_init( ) returns?

Common Bug II: Memory Leak

- Since the memory allocated by operator new is unnamed, always keep track of it using a pointer variable.
- If you lose track of it, it will become inaccessible and there will be memory leak.
- When you leak a lot of memory, then the computer does not have enough memory to run your program $\Rightarrow$ runtime error.

Example: Memory Leak

```
int *ip = new int; *ip = 5;                          // first unnamed int
ip = new int; *ip = 8;          // last unnamed int can't be accessed anymore
```



Example: Memory Leak

Example: Memory Leak Too

```
void swap(Date& x, Date& y)
{
    Date* temp = new Date; *temp = x; x = y; y = *temp;
}

int main(void)
{
    Date a = { 2006 , 1 , 10 }; Date b = { 2005 , 9 , 1 };
    swap(a, b); return 0;
}
```

- The variable, Date* temp is a local variable in the function swap( ).
- Everytime when swap( ) is called, temp is automatically allocated on a stack.
- new Date returns an unnamed memory of size equal to sizeof(Date) from the heap.

Example: Memory Leak ..

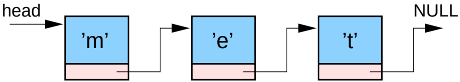
When swap( ) returns,

- the memory for local variables like temp will be deallocated automatically.
- However, the memory allocated by operator new remains until
  - operator delete is used to deallocate it.
  - the whole program finishes, the operating system will take back all memory dynamically allocated by the program that has not been deleted.

Question: What happens to the unnamed memory returned by $\boxed{\text{new Date}}$ when swap( ) returns back to main( )?

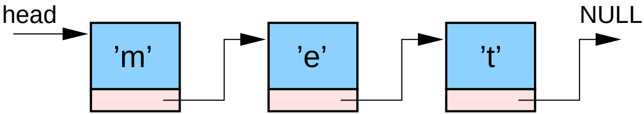
Part V

A Dynamic Data Structure: Linked List



What is a Linked List?

- A *list* is a linear sequence of objects.
- You may implement a list by an array. e.g. `int x[5];`
  - Advantage: array is an efficient data structure that works well with loops and recursion.
  - Disadvantage: size of the array is determined in advance.
- A linked list links objects together by pointers so that each object is pointing to the next object in the sequence (list).
  - Advantage: It is dynamic; it grows and shrinks to any size as you want at runtime.
  - Disadvantage:
    - requires additional memory for the linking pointers
    - takes more time to manipulate its items



A Typical C++ Linked List Definition

- Each object in a linked list is usually called a "node".
- The typical C++ definition for a node in a linked list is a struct (or later class):

```
struct ll_node
{
    <type> data;          // contains useful information
    ll_node* next;        // the link to the next node
};
```

- The first and the last node of a linked list always need special attention.
- For the last node, its next pointer is set to NULL to tell that it is the end of the linked list.
- We need a pointer variable, usually called head to point to the first node.
- Once you get the head of the linked list, you get the whole list!

Basic Operations of a Linked List

```
/* To create a node */
ll_node* p = new ll_node;

/* To access the data in a node */
cout << p->data;
cout << (*p).data;

/* To set up the head of a linked list */
ll_node* head = NULL;                    // an empty linked list
head = p;   // the linked list starts with the node pointed by p

/* To delete a node */
delete p;
```

Common Operations on a Linked List

- Common operations:
  - Create a new linked list.
  - Search data in the list.
  - Delete a node in the list.
  - Insert a new node in the list.
- For all these operation, again special attention is usually needed when the operation involves the first or the last node.

Example: LL-String — ll_cnode.h

Let's use a linked list (instead of an array) of characters to represent a string.

```
#include <iostream>                               /* File: ll_cnode.h */
using namespace std;

struct ll_cnode {
    char data;                                    // contains useful information
    ll_cnode* next;                               // the link to the next node
};

const char NULL_CHAR = '\0';
ll_cnode* ll_create(char);
ll_cnode* ll_create(const char [ ]);
int ll_length(const ll_cnode*);
void ll_print(const ll_cnode*);
ll_cnode* ll_search(ll_cnode*, char c);
void ll_insert(ll_cnode*&, char, unsigned);
void ll_delete(ll_cnode*&, char);
void ll_delete_all(ll_cnode*&);
```

Example: LL-String — ll_create.cpp

```
#include "ll_cnode.h"                                    /* File: ll_create.cpp */
// Create a ll_cnode and initialize its data
ll_cnode* ll_create(char c)
{
    ll_cnode* p = new ll_cnode; p->data = c; p->next = NULL; return p;
}

// Create a linked list of ll_cnodes with the contents of a char array
ll_cnode* ll_create(const char s[ ])
{
    if (s[0] == NULL_CHAR)          // Empty linked list due to empty C string
        return NULL;

    ll_cnode* head = ll_create(s[0]);                 // Special case with the head

    ll_cnode* p = head;                               // p is the working pointer
    for (int j = 1; s[j] != NULL_CHAR; ++j)
    {
        p->next = ll_create(s[j]);    // Link current ll_cnode to the new ll_cnode
        p = p->next;                            // p now points to the new ll_cnode
    }

    return head;        // The WHOLE linked list can be accessed from the head
}
```

Example: LL-String — ll_length.cpp, ll_print.cpp

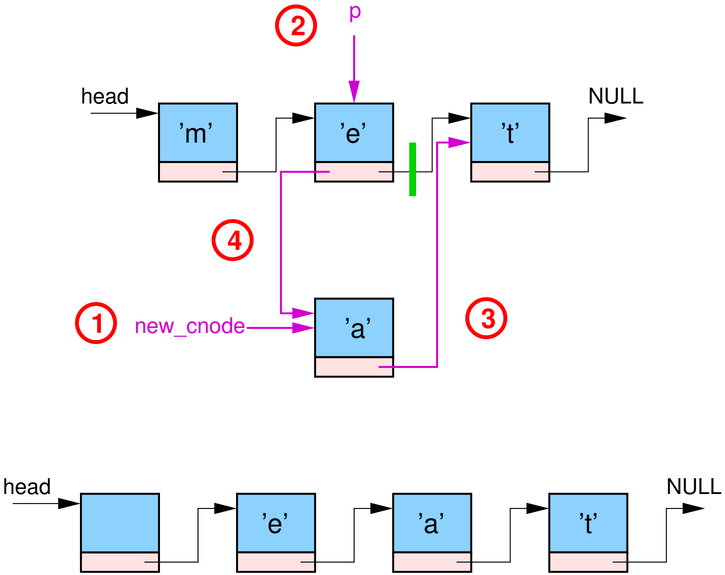
```
#include "ll_cnode.h"                          /* File: ll_length.cpp */

int ll_length(const ll_cnode* head)
{
    int length = 0;
    for (const ll_cnode* p = head; p != NULL; p = p->next)
        ++length;
    return length;
}
```

```
#include "ll_cnode.h"                          /* File: ll_print.cpp */

void ll_print(const ll_cnode* head)
{
    for (const ll_cnode* p = head; p != NULL; p = p->next)
        cout << p->data;
    cout << endl;
}
```

Example: LL-String — ll_search.cpp

```
#include "ll_cnode.h"                          /* File: ll_search.cpp */

// The returned pointer may be used to change the content
// of the found ll_cnode. Therefore, the return type
// should not be const ll_cnode*.

ll_cnode* ll_search(ll_cnode* head, char c)
{
    for (ll_cnode* p = head; p != NULL; p = p->next)
    {
        if (p->data == c)
            return p;
    }

    return NULL;
}
```

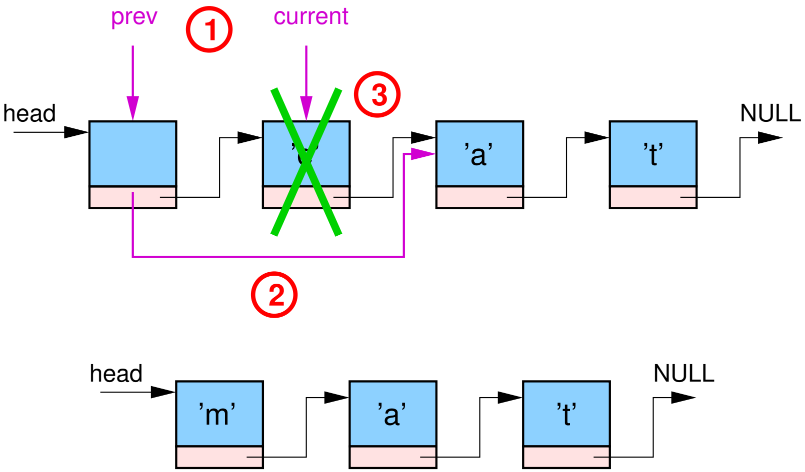
Example: LL-String — Insertion Algorithm



Example: LL-String — ll_insert.cpp

```
#include "ll_cnode.h"                                       /* File: ll_insert.cpp */
// To insert character c to the linked list so that after insertion,
// c is the n-th character (counted from zero) in the list.
// If n > current length, append to the end of the list.
void ll_insert(ll_cnode*& head, char c, unsigned n)
{
    ll_cnode* new_cnode = ll_create(c);    // STEP 1: Create the new ll_cnode
    if (n == 0 || head == NULL)             // Special case: insert at the beginning
    {
        new_cnode->next = head;
        head = new_cnode;
        return;
    }

    // STEP 2: Find the node after which the new node is to be added
    ll_cnode* p = head;
    for (int position = 0;
         position < n-1 && p->next != NULL;
         p = p->next, ++position)
         ;
    // STEP 3,4: Insert the new node between the found node and the next node
    new_cnode->next = p->next;                                          // STEP 3
    p->next = new_cnode;                                                // STEP 4
}
```

Example: LL-String — Deletion Algorithm



Example: LL-String — ll_delete.cpp

```
#include "ll_cnode.h"                                  /* File: ll_delete.cpp */
// To delete the character c from the linked list.
// Do nothing if the character cannot be found.
void ll_delete(ll_cnode*& head, char c)
{
    ll_cnode* prev = NULL;                          // Point to previous ll_cnode
    ll_cnode* current = head;                        // Point to current ll_cnode

    // STEP 1: Find the item to be deleted
    while (current != NULL && current->data != c)
    {
        prev = current;                                // Advance both pointers
        current = current->next;
    }

    if (current != NULL)                                        // Data is found
    {                                               // STEP 2: Bypass the found item
        if (current == head)                        // Special case: delete the first item
            head = head->next;
        else
            prev->next = current->next;
        delete current;      // STEP 3: Free up the memory of the deleted item
    }
}
```

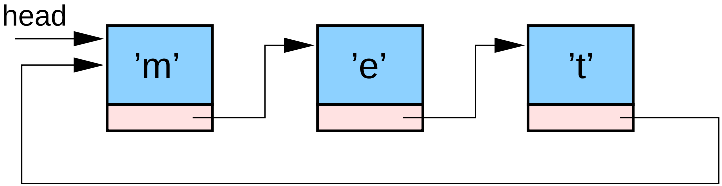
Example: LL-String — ll_delete_all.cpp

```
#include "ll_cnode.h"                     /* File: ll_delete_all.cpp */

// To delete the WHOLE linked list, given its head.
void ll_delete_all(ll_cnode*& head)
{
    if (head == NULL)                 // An empty list; nothing to delete
        return;

    ll_delete_all(head->next);  // STEP 1: First delete the remaining nodes

    // For debugging: this shows you what are deleting
    cout << "deleting " << head->data << endl;
    delete head;                // STEP 2: Then delete the current nodes

    head = NULL;                // STEP 3: To play safe, reset head to NULL
}
```

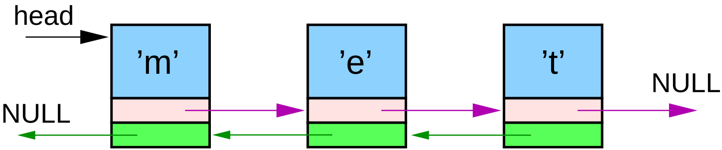
Example: LL-String — ll_test.cpp

```
#include "ll_cnode.h"                                   /* File: ll_test.cpp */
int main(void)
{
    ll_cnode* ll_string = ll_create("met");
    cout << "length of ll_string = " << ll_length(ll_string) << endl;
    ll_print(ll_string);
    ll_print(ll_search(ll_string, 'e'));

    cout << endl << "After inserting 'a'" << endl;
    ll_insert(ll_string, 'a', 2); ll_print(ll_string);

    cout << endl << "After deleting 'e'" << endl;
    ll_delete(ll_string, 'e'); ll_print(ll_string);
    cout << endl << "After deleting 'm'" << endl;
    ll_delete(ll_string, 'm'); ll_print(ll_string);
    cout << endl << "After inserting 'e'" << endl;
    ll_insert(ll_string, 'e', 9); ll_print(ll_string);
    cout << endl << "After deleting 't'" << endl;
    ll_delete(ll_string, 't'); ll_print(ll_string);
    cout << endl << "After deleting 'e'" << endl;
    ll_delete(ll_string, 'e'); ll_print(ll_string);
    cout << endl << "After deleting 'a'" << endl;
    ll_delete(ll_string, 'a'); ll_print(ll_string);

    cout << endl << "After deleting 'z'" << endl;
    ll_delete(ll_string, 'z'); ll_print(ll_string);
    cout << endl << "After inserting 'h'" << endl;
    ll_insert(ll_string, 'h', 9); ll_print(ll_string);
    cout << endl << "After inserting 'o'" << endl;
    ll_insert(ll_string, 'o', 0); ll_print(ll_string);

    ll_delete_all(ll_string);
    return 0;
}
```

- Circular Linked List



- Doubly Linked List

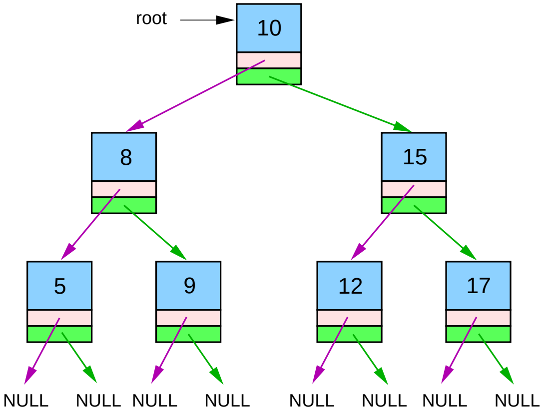


Part VI

Another Dynamic Data Structure: (Binary) Tree

What is a Binary Tree?

- An important dynamic data structure in CS is *tree*.
- CS's tree actually looks like an inverted physical tree.
- In particular, any node of a binary tree has 2 sub-trees (children): left sub-tree (child) and right sub-tree (child).



```
#include <iostream>                              /* File: btree.h */
using namespace std;

struct btree_node                                // A node in a binary tree
{
    int data;
    btree_node* left;                   // Left sub-tree or called left child
    btree_node* right;                // Right sub-tree or called right child
};


// Function declarations
btree_node* create_btree_node(int data);
void delete_btree(btree_node*& tree) ;
void print_btree(const btree_node* tree, int depth = 0);
```

Example: Binary Tree — btree-main.cpp

```
#include "btree.h"                              /* File: btree-main.cpp */

int main(void)
{
    btree_node* root = create_btree_node(10);       // Create root node

    root->left = create_btree_node(8);              // Create the left sub-tree
    root->left->left = create_btree_node(5);
    root->left->right = create_btree_node(9);

    root->right = create_btree_node(15);            // Create the right sub-tree
    root->right->left = create_btree_node(12);
    root->right->right = create_btree_node(17);

    print_btree(root);                              // Print the resulting tree

    delete_btree(root->left);                       // Delete the left sub-tree
    cout << "\n\n\n"; print_btree(root);            // Print the resulting tree
    return 0;
}
```

Example: Binary Tree — btree-create-delete.cpp

```
#include "btree.h"                      /* File: btree-create-delete.cpp */

btree_node* create_btree_node(int data)
{
    btree_node* node = new btree_node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

void delete_btree(btree_node*& tree)                      // By recursion
{
    if (tree == NULL) return;                             // Base case

    delete_btree(tree->left);            // Recursion on the left subtree
    delete_btree(tree->right);          // Recursion on the right subtree
    delete tree;
    tree = NULL;
}
```

```
#include "btree.h"                                   /* File: btree-print.cpp */

void print_btree(const btree_node* tree, int depth)
{
    if (tree == NULL)                                // Base case
        return;

    print_btree(tree->right, depth+1);               // Recursion: right subtree

    for (int j = 0; j < depth; j++)                  // Print the node data
        cout << '\t';
    cout << tree->data << endl;

    print_btree(tree->left, depth+1);                // Recursion: left subtree
}
```

Part VII

Array as a Pointer

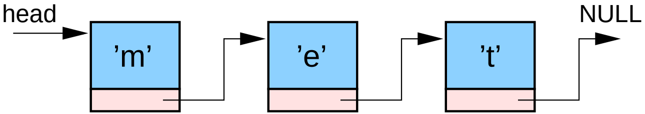
Pointer Arithmetic

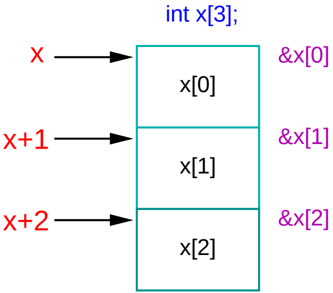
- A pointer variable supports 2 arithmetic operations: $+$, -.
- If you have $\boxed{\text{<type> x; <type>* xp} = \text{\&x;}}$, then
  - $\boxed{\text{xp} + \text{N}}$ $==$ &x $+$ sizeof(<type>) $\times$ N.
  - $\boxed{\text{xp - N}}$ $==$ &x - sizeof(<type>) $\times$ N.
- The result of pointer arithmetic should be a valid address, otherwise, dereferencing it may lead to segmentation fault!

Example: Pointer Arithmetic

```
#include <iostream>                              /* File: pointer-math.cpp */
using namespace std;

int main(void)
{
    double x = 2.3;                                     // double is 8-byte
    double* xp = &x;                                    // xp points to x
    cout << &x << endl << xp + 2 << endl << xp - 2 << endl;

    // Nothing disallows you from assigning an integer value
    // to a pointer variable. Hexadecimal numbers start with 0x.
    int* yp = reinterpret_cast<int*>(0x14);
    cout << yp + 1 << endl << yp - 1 << endl;

    // Since addresses around 0x14 may not be accessible to you
    // Dereferencing them usually leads to runtime error
    cout << *(yp + 1) << endl << *(yp - 1) << endl;
    return 0;
}
```

Array Name is Actually a const Pointer!





- Just like that the head of a linked list is a pointer to the first element of the list, the array identifier can also be interpreted as a pointer to the first array element.
- In fact, the array identifier is a const pointer.
- Thus, the variable x in $\boxed{\text{int x[3];}}$ has the type $\boxed{\text{int* const}}$.

- Any pointer pointing to an array can be used to access all elements of the array instead of the original array identifier.

```
#include <iostream>            /* File: array-by-another-pointer.cpp */
using namespace std;

int main(void)
{
    int x[ ] = { 11, 22, 33, 44 };
    int* y = x;   // Both y and x point to the 1st element of array

    // Modify the array through pointer y
    for (int j = 0; j < sizeof(x)/sizeof(int); ++j)
        y[j] += 100;

    // Print the array through pointer x
    for (int j = 0; j < sizeof(x)/sizeof(int); ++j)
        cout << x[j] << endl;
    return 0;
}
```

Access Array Items by Pointer Arithmetic & Dereferencing

- Using pointer arithmetic, you may "move" a pointer to point to any array element.
- Dereferencing a pointer to an array element then obtains the element — and you can use it as either lvalue or rvalue.
- Again, if `int x[ ] = {11,22,33}; int* xp = x;`, then we have

Element Address	Element Value
xp == x == &x[0]	*xp == *x == x[0] == 11
xp+1 == x+1 == &x[1]	*(xp+1) == *(x+1) == x[1] == 22
xp+2 == x+2 == &x[2]	*(xp+2) == *(x+2) == x[2] == 33

And by definition, numerically, we have &x == x == &x[0] .

Example: Print an Array using Pointer

```
#include <iostream>                    /* File: print-array-by-pointer.cpp */
using namespace std;
int main(void)
{
    int x[ ] = { 11, 22, 33, 44 };
    for (int* xp = x, j = 0; j < sizeof(x)/sizeof(int); ++j, ++xp)
        cout << *xp << endl;
    return 0;
}
```

```
#include <iostream>               /* File: print-char-array-by-pointer.cpp */
using namespace std;
int main(void)
{
    char s[ ] = "hkust";
    for (const char* sp = s; *sp != '\0'; ++sp)
        cout << *sp << endl;
    return 0;
}
```

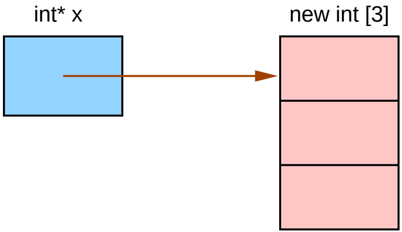
Creation of Dynamic Array: Operator new Again

Syntax: new a Dynamic Array

<type>* <pointer-variable> =
    new <type> [<integer-expression>] ;

Examples: Use of the new Operator

```
int array_size;                          // Unknown till runtime
cin >> array_size;
int* x = new int [array_size];
for (int j = 0; j < array_size; ++j)
    x[j] = j;
```



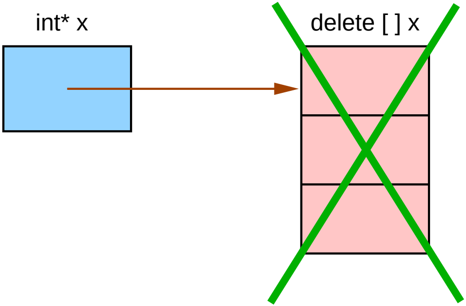
Destruction of Dynamic Array: Operator delete Again

Syntax: delete a Dynamic Array

delete [ ] <pointer-variable> ;

Examples: Use of the new Operator

```
delete [ ] x;       // x is now a dangling pointer
x = NULL;           // x is now a NULL pointer
```



Example: Dynamic 1D Array

```
#include <iostream>                                  /* File: dynamic-point-array.cpp */
#include "point.h"
using namespace std;

int main(void)
{
    void print_distance(const Point*, const Point*);
    int num_points;
    cout << "Enter the number of points :  "; cin >> num_points;

    Point* point = new Point [num_points]; // Allocate a dynamic array of points
    for (int j = 0; j < num_points; ++j)                    // Input the points
    {
        cout << "Enter the x & y coordinates of point #" << j << " :  ";
        cin >> point[j].x >> point[j].y;
    }

    for (int i = 0; i < num_points; ++i)  // Compute distance between 2 points
        for (int j = 0; j < num_points; ++j)
            if (i < j)
                print_distance(point+i, point+j);

    delete [ ] point;                  // Deallocate the dynamic array of points
    return 0;
} /* g++ dynamic-point-array.cpp point-distance.cpp */
```

Example: Dynamic 1D Array ..

```
#include <iostream>                              /* File: point-distance.cpp */
#include <cmath>
#include "point.h"
using namespace std;

double euclidean_distance(const Point* p1, const Point* p2)
{
    double x_diff = p1->x - p2->x;
    double y_diff = p1->y - p2->y;
    return sqrt(x_diff*x_diff + y_diff*y_diff);
}

void print_point(const Point* p)
{
    cout << '(' << p->x << ", " << p->y << ')';
}

void print_distance(const Point* p1, const Point* p2)
{
    cout << "Distance between "; print_point(p1);
    cout << " and "; print_point(p2);
    cout << " is " << euclidean_distance(p1, p2) << endl;
}
```

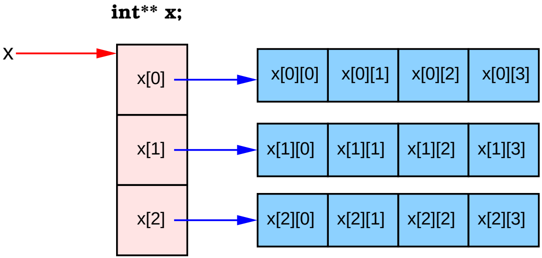
Example: C String as Char Pointer

```
#include <iostream>                               /* File: palindrome.cpp */
using namespace std;

bool palindrome(const char* first, const char* last)
{
    if (first >= last)
        return true;
    else if (*first != *last)
        return false;
    else
        return palindrome(first+1, last-1);
}

int main(void)
{
    const int MAX_LINE_LEN = 255;
    char s[MAX_LINE_LEN+1];

    while (cin.getline(s, MAX_LINE_LEN+1, '\n'))
        cout << boolalpha << palindrome(s, s+strlen(s)-1) << endl;
    return 0;
}
```

Part VIII

Multi-dimensional Array and Pointer

Dynamic Allocation of a 2D Array



- To create a 2D int array with M rows and N columns at runtime:
  1. Allocate a 1D array of M int* (int pointers).
  2. For each of the M elements, create another 1D array of N int (integers), and set the former to point to the latter.

Question: Can you generalize this to 3D, 4D, . . ., arrays?

Example: Operations of a Dynamic 2D Array

```
#include <iostream>                    /* File: 2d-dynamic-array-main.cpp */
using namespace std;

int** create_matrix(int, int);
void print_matrix(const int* const*, int, int);
void delete_matrix(int**, int, int);

int main(void)
{
    int num_rows, num_columns;
    cout << "Enter #rows followed by #columns:  ";
    cin >> num_rows >> num_columns;
    int** matrix = create_matrix(num_rows, num_columns);

    // Dynamic array elements can be accessed like static array elements
    for (int j = 0; j < num_rows; ++j)
        for (int k = 0; k < num_columns; ++k)
            matrix[j][k] = 10*(j+1) + (k+1);

    print_matrix(matrix, num_rows, num_columns);
    delete_matrix(matrix, num_rows, num_columns);
    matrix = NULL;                          // Avoid dangling pointer
    return 0;
}  /* g++ 2d-dynamic-array-main.cpp 2d-dynamic-array-functions.cpp */
```

```
#include <iostream>                          /* File: 2d-dynamic-array-functions.cpp */
using namespace std;

int** create_matrix(int num_rows, int num_columns) {
    int **x = new int* [num_rows];                                       // STEP 1
    for (int j = 0; j < num_rows; ++j)                                   // STEP 2
        x[j] = new int [num_columns];
    return x;
}

void print_matrix(const int* const* x, int num_rows, int num_columns) {
    for (int j = 0; j < num_rows; ++j)
    {
        for (int k = 0; k < num_columns; ++k)
            cout << x[j][k] << '\t';
        cout << endl;
    }
}

void delete_matrix(int** x, int num_rows, int num_columns) {
    for (int j = 0; j < num_rows; ++j)            // Delete is done in reverse order
        delete [ ] x[j];                           // (compared with its creation)
    delete [ ] x;
}
```

```
#include <iostream>             /* File: 2d-dynamic-array-and-pointer.cpp */
using namespace std;

int main(void)
{
    // Dynamically create an array with 3 rows, 4 columns
    int **x = new int* [3];                                        // STEP 1
    for (int j = 0; j < 3; j++)                                    // STEP 2
        x[j] = new int [4];

    cout << endl << "Info about x:" << endl;
    cout << "sizeof(x) :\t" << sizeof(x) << endl << endl;
    cout << "x" << "\t\t" << "&x[0]" << "\t\t" << "&x[0][0]" << endl;
    cout << x << '\t' << &x[0] << '\t' << &x[0][0] << endl << endl;

    cout << "&x[j]" << "\t\t" << "x[j]" << "\t\t"
         << "&x[j][0]" << '\t' << "x+j" << endl;
    for (int j = 0; j < 3; j++)
        cout << &x[j] << '\t' << x[j] << '\t' << &x[j][0] << '\t' << x+j << endl;
    return 0;
}
```

```
Info about x:
sizeof(x) :     8

x               &x[0]           &x[0][0]
0x14ea5010      0x14ea5010      0x14ea5030

&x[j]           x[j]            &x[j][0]        x+j
0x14ea5010      0x14ea5030      0x14ea5030      0x14ea5010
0x14ea5018      0x14ea5050      0x14ea5050      0x14ea5018
0x14ea5020      0x14ea5070      0x14ea5070      0x14ea5020
```

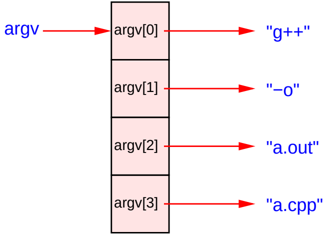
Notice that, numerically, we have

- x == &x[0] != &x[0][0]
  ⇒ x points to x[0] (and not x[0][0] as in static 2D array)
- &x[j] == x+j
  ⇒ a proof of the pointer arithmetic.
- x[j] == &x[j][0]
  ⇒ x[j] points to the first element of the *j*th row.

- Up to now, you write the main function header as
  int main( ) or int main(void).
- In fact, the general form of the main function allows variable number of arguments (overloaded function).

```
int main(int argc, char** argv)
int main(int argc, char* argv[ ])
```

- argc gives the actual number of arguments.
- argv is an array of char*, each pointing to a character string.
- e.g. g++ -o a.out a.cpp calls the main function of the g++ program with 3 additional commandline arguments. Thus, argc = 4, and



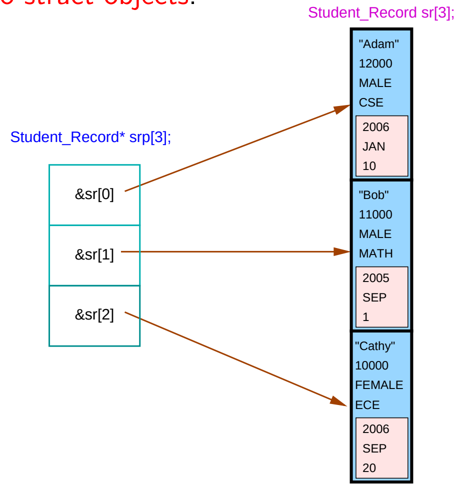
```
#include <iostream>                /* File: 2d-dynamic-array-main-with-argv.cpp */
using namespace std;
int** create_matrix(int, int);
void print_matrix(const int* const*, int, int);
void delete_matrix(int**, int, int);

int main(int argc, char** argv)
{
    if (argc != 3)
    { cerr << "Usage:  " << argv[0] << " #rows #columns" << endl; return -1; }

    int num_rows = atoi(argv[1]);
    int num_columns = atoi(argv[2]);
    int** matrix = create_matrix(num_rows, num_columns);

    // Dynamic array elements can be accessed like static array elements
    for (int j = 0; j < num_rows; ++j)
        for (int k = 0; k < num_columns; ++k)
            matrix[j][k] = 10*(j+1) + (k+1);

    print_matrix(matrix, num_rows, num_columns);
    delete_matrix(matrix, num_rows, num_columns);
    matrix = NULL;                               // Avoid dangling pointer
    return 0;
}  /* g++ 2d-dynamic-array-main-with-argv.cpp 2d-dynamic-array-functions.cpp */
```

Part IX

Further Reading: Array of Pointers to Structures

Array of Pointers to Structures

- You may create an array of basic data types as well as user-defined data types, or pointers to them.
- Thus, you may have an array of struct objects, or an array of pointers to struct objects.



Example: (Previously) Sort by Struct Objects Themselves

```
#include "student-record.h"                    /* File: sort-student-record.cpp */
#include "student-record-extern.h"

int main(void)
{
    Student_Record sr[ ] = {
        { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
        { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
        { "Cathy", 10000, 'F', ECE, { 2006 , 8 , 20 } } };

    Date d;                                       // Modify the 3rd record
    set_date(&d, 1980, 12, 25);
    set_student_record(&sr[2], "Jane", 18000, 'F', CSE, &d);

    sort_3SR_by_id(sr);
    for (int j = 0; j < sizeof(sr)/sizeof(Student_Record); j++)
        print_student_record(&sr[j]);
    return 0;
}
/* g++ -o sort-sr sort-student-record.cpp student-record-functions.cpp
   student-record-swap.cpp */
```

Advantage of Indirect Addressing

- During a sorting procedure, in general, many array items are swapped.
- When 2 items are swapped, 3 copy actions are required.
- When the array items are big — say, 1MB — objects, the copying actions may take substantial amount of computation and time.
- A common solution is to make use of indirect addressing and to sort using the pointers to the objects instead.
- The size of pointers is fixed, independent of the objects they point to. For a 32-bit CPU, it is 4 bytes; for a 64-bit CPU, it is 8 bytes.
- When 2 items are sorted and swapped by their pointers, the 3 copy actions involves only copying 4-byte pointers (for 32-bit CPU and 8-byte pointers for 64-bit CPU) which are independent of the size of items they point to.

Example: Sort by Pointers to Struct Objects

```
#include "student-record.h"                /* File: sort-student-record-ptr.cpp */
void swap_SR_ptr(Student_Record*&, Student_Record*&);
void print_student_record(const Student_Record*);

void sort_3SR_by_id_by_ptr(Student_Record* srp[ ])
{
    if (srp[0]->id > srp[1]->id) swap_SR_ptr(srp[0], srp[1]);
    if (srp[0]->id > srp[2]->id) swap_SR_ptr(srp[0], srp[2]);
    if (srp[1]->id > srp[2]->id) swap_SR_ptr(srp[1], srp[2]);
}

int main(void)
{
    Student_Record sr[ ] = {
        { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
        { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
        { "Cathy", 10000, 'F', ECE, { 2009 , 6 , 20 } } };

    Student_Record* srp[ ] = { &sr[0], &sr[1], &sr[2] };        // Array of pointers
    sort_3SR_by_id_by_ptr(srp);

    for (int j = 0; j < sizeof(srp)/sizeof(Student_Record*); ++j)
        print_student_record(srp[j]);
    return 0;
}   /* g++ sort-student-record-ptr.cpp student-record-ptr-functions.cpp */
```

Example: Sort by Pointers to Struct Objects ..

```
#include <iostream>                    /* File: student-record-ptr-functions.cpp */
#include "student-record.h"
using namespace std;

// Swap 2 Student_Record's by their pointers
void swap_SR_ptr(Student_Record*& srp1, Student_Record*& srp2)
{
    Student_Record* temp = srp1; srp1 = srp2; srp2 = temp;
}

void print_date(const Date* date)
{
    cout << date->year << '/' << date->month << '/' << date->day << endl;
}

void print_student_record(const Student_Record* x) {
    cout << endl;
    cout.width(12); cout << "name:   " << x->name << endl;
    cout.width(12); cout << "id:   " << x->id << endl;
    cout.width(12); cout << "gender:   " << x->gender << endl;
    cout.width(12); cout << "dept:   " << dept_name[x->dept] << endl;
    cout.width(12); cout << "entry date:   "; print_date(&x->entry);
}
```

Another Way of Implementing Pointer by Index

- The principle of "sort-by-pointers" is that the actual objects in an array do not move. Instead, their pointers move to indicate their positions during and after sorting.
- Before we have C++ pointers, one may implement the same concept by using a separate array of object indices.
- In a similar fashion, one sort the actual objects by manipulating their indices (which are conceptually equivalent to the pointers).

Example: Sort by Indices to Struct Objects

```
#include "student-record.h"          /* File: sort-student-record-by-index.cpp */
void swap_SR_index(int&, int&);
void print_student_record(const Student_Record&);

void sort_3SR_by_id_by_index(Student_Record sr[ ], int index[ ])
{
    if (sr[index[0]].id > sr[index[1]].id) swap_SR_index(index[0], index[1]);
    if (sr[index[0]].id > sr[index[2]].id) swap_SR_index(index[0], index[2]);
    if (sr[index[1]].id > sr[index[2]].id) swap_SR_index(index[1], index[2]);
}

int main(void)
{
    Student_Record sr[ ] = {
        { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
        { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
        { "Cathy", 10000, 'F', ECE, { 2009 , 6 , 20 } } };

    int index[ ] = { 0, 1, 2 };           // Array of indices of student records
    sort_3SR_by_id_by_index(sr, index);

    for (int j = 0; j < sizeof(index)/sizeof(int); ++j)
        print_student_record(sr[index[j]]);
    return 0;
}   // g++ sort-student-record-by-index.cpp student-record-by-index-functions.cpp
```

```
#include <iostream>            /* File: student-record-by-index-functions.cpp */
#include "student-record.h"
using namespace std;

// Swap 2 Student_Record's by their indices
void swap_SR_index(int& index1, int& index2)
{
    int temp = index1; index1 = index2; index2 = temp;
}

void print_date(const Date& date)
{
    cout << date.year << '/' << date.month << '/' << date.day << endl;
}

void print_student_record(const Student_Record& x) {
    cout << endl;
    cout.width(12); cout << "name:   " << x.name << endl;
    cout.width(12); cout << "id:   " << x.id << endl;
    cout.width(12); cout << "gender:   " << x.gender << endl;
    cout.width(12); cout << "dept:   " << dept_name[x.dept] << endl;
    cout.width(12); cout << "entry date:   "; print_date(x.entry);
}
```