COMP 2012H Honors Object-Oriented Programming and Data Structures

Topic 8: C++ Pointers & Dynamic Data

Dr. Desmond Tsoi

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

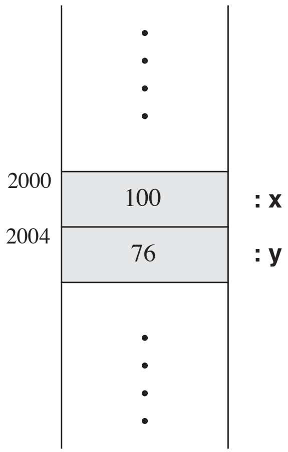
Part I

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.



lvalue & rvalue

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
```

lvalue & rvalue: Return-by-Reference vs. Return-by-Value

++x: the pre-increment operator

1. requires x to be passed-by-reference
2. modify x by incrementing it by 1
3. returns x (with its new value) by reference
4. the returned x is an lvalue

x++: the post-increment operator

1. requires x to be passed-by-reference
2. saves the current value of x in some temporary local variable
3. modify x by incrementing it by 1
4. returns the old value of x in the local variable by value
5. the returned x is an rvalue

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()
{
    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

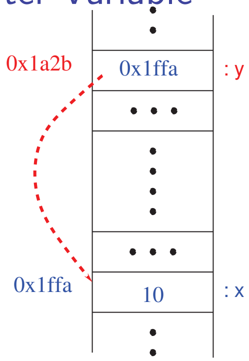
```
#include <iostream>     /* File: fcn-var-addr.cpp */
using namespace std;

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()
{
    int x = 10, y = 20;
    cout << endl << "Inside main()" << 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.

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

int main() {
    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()
{
    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 content from the address stored in y, put it into z
    cout << endl;
    cout << "z = " << z << '\t' << "address of z = " << &z << endl;
    cout << "y = " << y << '\t' << "*y = " << *y << endl;
    return 0;
}
```


Example: Pointer Manipulation

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

int main()
{
    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()
{
    char   c = 'A';   char* pc = &c;
    short  s = 5;    short* ps = &s;
    int    i = 10;     int* pi = &i;
    double d = 5.6; 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 ⇒ function pointer! (discussed later)

Example: Pointer to Pointer to Pointer ...

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

int main()
{
    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

```
#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()
{
    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()
{
    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
```

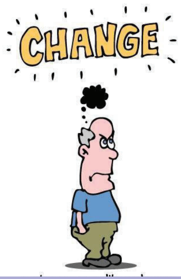
Pointer to const Objects

Syntax: Definition of Pointer to a const Object

const <type>* <pointer variable>;

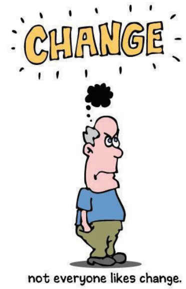
Example: Pointer to const Object

```
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
```



Pointer to const Objects ..

- 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!



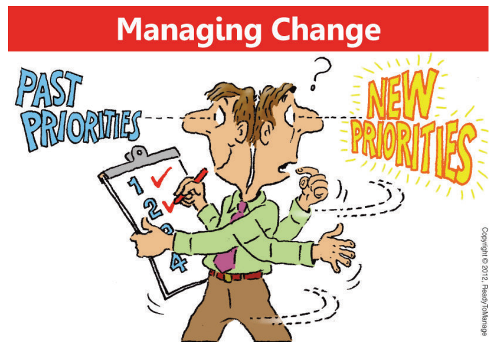
Quiz: (const) Pointer to (const) Objects

Can you tell the differences among the following?

- `int* p;`
- `const int* p;`
- `int* const p;`
- `const int* const p;`

PBR = PBV + Pointer

- The programming language C only has 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

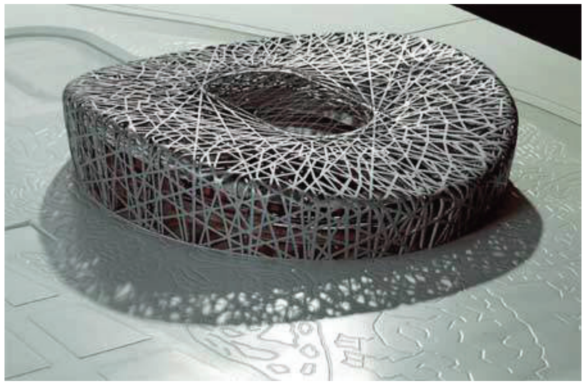
```
#include <iostream>      /* File: pbv-pointer.cpp */
using namespace std;

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()
{
    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
- Dynamic object creation/deletion
- Advanced uses that will be covered in this course later:
  - 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



Pointer to struct and the → Operator

- 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 contains garbage
Point* ap = &a;          // Now ap points to a

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

  2. Directly use the → operator.

```
Point a;                 // a contains 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;

// To compute and print the Euclidean distance between 2 points
void print_distance(const Point*, const Point*);

int main()   /* 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 */
```


Example: Euclidean Distance Again — 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, 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()
{
    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 I

```
#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);
}
```


Example: student-record-functions.cpp Again II

```
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

Static Objects

Example: Static Objects

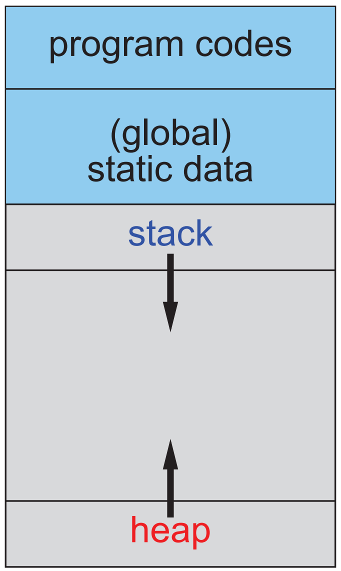
```
float a = 2.3;           // Global float variable

int main()
{
    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



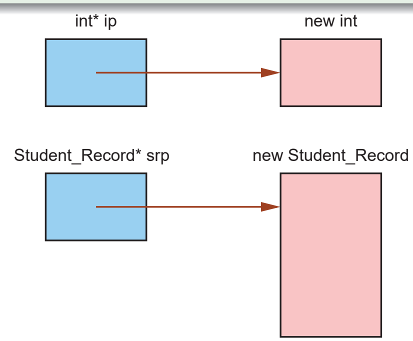
Dynamic Memory Allocation: Operator new

Syntax: Dynamic Memory Allocation Using new

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

Examples: Use of the new Operator

```
int* ip = new int;
*ip = 5;
Date d20010101 = { 2001, 1 , 1 };
Student_Record* srp = new Student_Record;
set_student_record(*srp, "Chris", 100, 'M', CSE, d20010101);
```



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.

Dynamic Memory Allocation: Operator new ...

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.

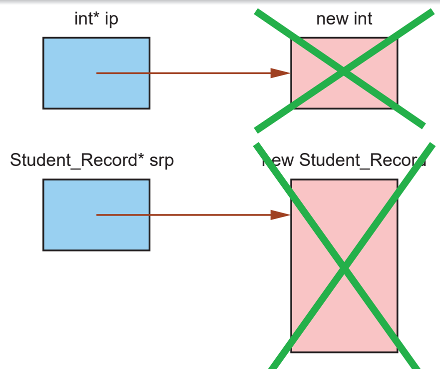
Dynamic Memory Deallocation: Operator delete

Syntax: Dynamic Memory Deallocation Using delete

delete <pointer-variable>;

Examples: Use of the delete Operator

```
delete ip;              // ip is now a dangling pointer
ip = nullptr;           // ip is now a null pointer
delete srp;             // srp is now a dangling pointer
srp = nullptr;          // 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 nullptr.
- nullptr is a new keyword in C++11 and is used to indicate a pointer that has not been set to point to something useful.
- In the past, a null pointer is represented by NULL or 0.
- Good practices:
  1. Always initialize a pointer to nullptr when defining a pointer variable.
  2. Always check whether a pointer is a nullptr before using it.

Common Bug I: Dangling Pointer — Case 2

Example: Dangling Pointer

```
int* create_and_init(int value) /* File: dangling-pointer.cpp */
{
    int x = value;      // x is a local variable
    int* p = &x;        // p is also a local variable
    return p;
}

int main()
{
    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?

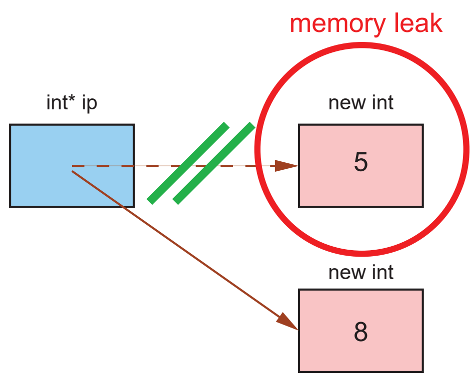
Common Bug II: Memory Leak

- Memory leak occurs when dynamically allocated memory that is no longer needed is not released.
- 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 ⇒ runtime error.

Common Bug II: Memory Leak ..

Example: Memory Leak

```
int* ip = new int; // First unnamed int
*ip = 5;
ip = new int;      // Last unnamed int is lost
*ip = 8;
```



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()
{
    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 new Date when swap( ) returns back to main( )?

Part V

Array as a Pointer

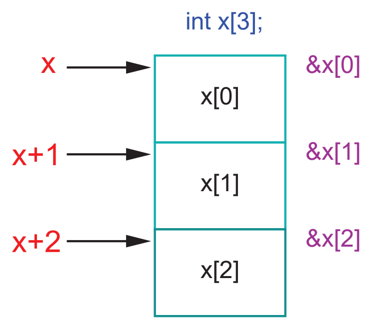
Pointer Arithmetic

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

Example: Pointer Arithmetic

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

int main()
{
    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 Can be Treated as a const Pointer!



- In fact, the array identifier can be treated as a const pointer to the first array element.
- Thus, the variable x in int x[3]; from the pointer perspective, is like int* const.

Access Array Items by Another Pointer

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

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

int main()
{
    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()
{
    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()
{
    char s[] = "hkust";
    for (const char* sp = s; *sp != '\0'; ++sp)
        cout << *sp << endl;
    return 0;
}
```

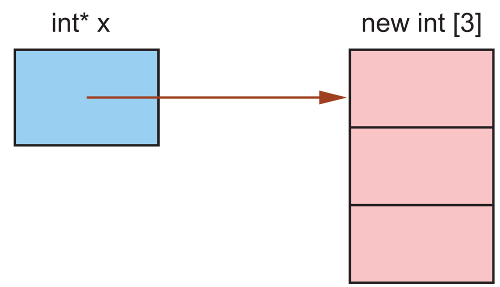

Creation of Dynamic Array: Operator new Again

Syntax: new a Dynamic Array

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

Examples: Use of the new Operator

```
int array_size; cin >> array_size; // Unknown till runtime
int* x = new int [array_size];
for (int j = 0; j < array_size; ++j)
    x[j] = j;   // Actually a pointer but treated like an array
```



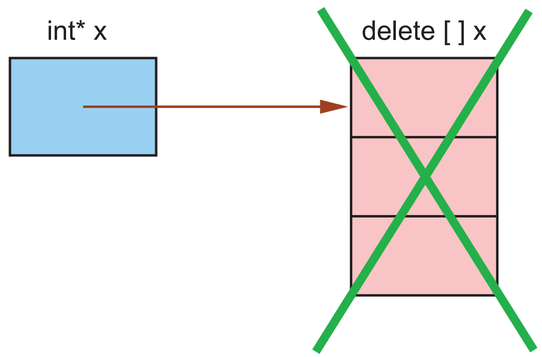
Destruction of Dynamic Array: Operator delete Again

Syntax: delete a Dynamic Array

delete [ ] <pointer-variable> ;

Examples: Use of the new Operator

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



Example: Dynamic 1D Array

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

int main()
{
    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]; // 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 = i+1; j < num_points; ++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 */
```

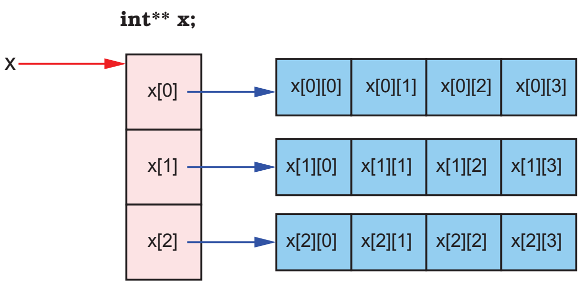
Example: Dynamic 1D Array ..

```
#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, 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;
}
```


Part VI

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()
{
    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 = nullptr;       // Avoid dangling pointer
    return 0;
} /* g++ 2d-dynamic-array-main.cpp 2d-dynamic-array-functions.cpp */
```

Example: Operations of a Dynamic 2D Array ..

```
#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;
}
```


Example: Relation between Dynamic 2D Array & Pointer

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

int main()
{
    // 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;
}
```


Example: Relation between Dynamic 2D Array & Pointer ..

```
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

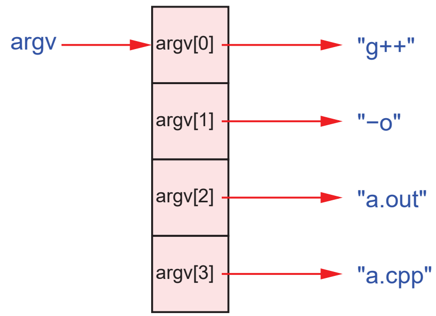
- x == &x[0] != &x[0][0]
  ⇒ x points to x[0] (and not x[0][0] as in static 2D array)
- &x[j] == x+j
  ⇒ a proof of the pointer arithmetic.
- x[j] == &x[j][0]
  ⇒ x[j] points to the first element of the *j*th row.

main( ) Function Arguments

- 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



Example: Operations of a Dynamic 2D Array using argv

```
#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 = nullptr; // Avoid dangling pointer
    return 0;
} /* g++ 2d-dynamic-array-main-with-argv.cpp 2d-dynamic-array-functions.cpp */
```

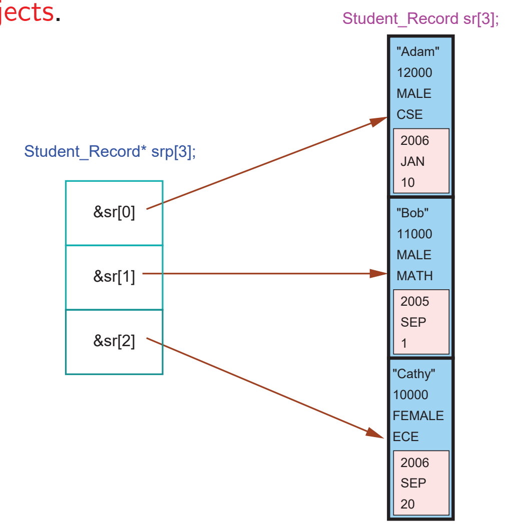

That's all!

Any questions?

Further Reading

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()
{
    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 involve 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()
{
    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()
{
    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
```


Example: Sort by Indices to Struct Objects ..

```
#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);
}
```