

Programming with C++

COMP2011: Pointers & Dynamic Data

Cecia Chan

Cindy Li

Brian Mak

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



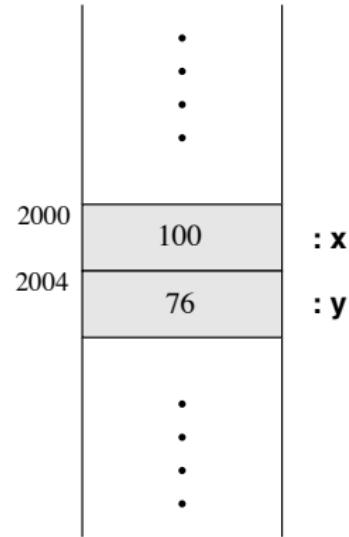
Part I

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. $x = 100;$ x is a variable and 100 is a literal constant.



Example: Ivalue and rvalue

```
x = x + 1;
```

- A variable has **dual** roles. Depending on where it appears in the program, it can represent an
 - **Ivalue**: **location** of the memory storage (**read-write**)
 - **rvalue**: **value** in the storage (**read-only**)
- They are so called because a variable represents an **Ivalue** (**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
```

$\textcolor{red}{++x}$: the pre-increment operator

- ➊ requires x to be **passed-by-reference**
- ➋ modify x by incrementing it by 1
- ➌ returns x (with its new value) by **reference**
- ➍ the returned x is an **lvalue**

$\textcolor{red}{x++}$: the post-increment operator

- ➊ requires x to be **passed-by-reference**
- ➋ saves the current value of x in some **temporary local variable**
- ➌ modify x by incrementing it by 1
- ➍ returns the old value of x in the local variable by **value**
- ➎ the returned value is an **rvalue**

Analogy: Website Has an **lvalue** and an **rvalue** Too!

The screenshot shows the homepage of the Department of Computer Science at The Hong Kong University of Science and Technology. A red arrow points to the URL bar with the address <https://www.cse.ust.hk>, labeled "lvalue of CSE website (address)". A blue arrow points to the main content area of the page, labeled "rvalue of CSE website (content)". The page features the university's logo and the department's name, along with navigation links for Intranet, School of Engineering, About, People, Research, Academics, Admissions, and a search bar. A large aerial photograph of the university campus is the central visual element, with a zoom control overlay. To the right, a sidebar titled "Quick Links for" lists Undergraduates, Postgraduates, Faculty & Staff, Alumni, Job Seekers, and Employers & Industry Partners.

Get the Address by the Reference Operator &

Syntax: Get the Address of a Variable
& <variable>

```
1 #include <iostream>      /* File: var-addr.cpp */
2 using namespace std;
3
4 int main()
5 {
6     int x = 10, y = 20;
7     short a = 9, b = 99;
8
9     cout << "x = " << x << '\t' << "address of x = " << &x << endl;
10    cout << "y = " << y << '\t' << "address of y = " << &y << endl;
11    cout << "a = " << a << '\t' << "address of a = " << &a << endl;
12    cout << "b = " << b << '\t' << "address of b = " << &b << endl;
13    return 0;
14 }
```

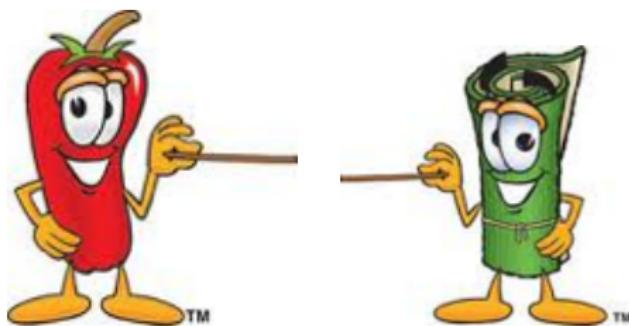
Example: Address of Formal Parameters

```
1 #include <iostream>      /* File: fcn-var-addr.cpp */
2 using namespace std;
3
4 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;
}
13
14 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;
}
```

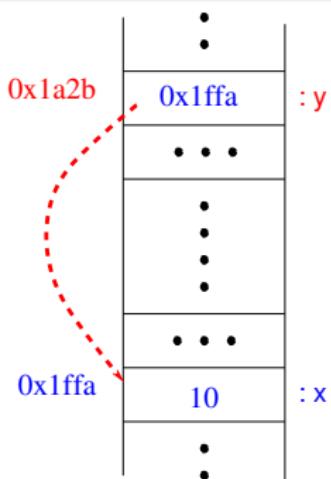
Question: Can you see the difference between PBV and PBR?

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>

```
1 #include <iostream>      /* File: pointer-deref.cpp */
2 using namespace std;
3
4 int main()
5 {
6     int x = 10, z = 20;
7     int* y = &x; // y now contains the address of x
8     cout << "x = " << x << '\t' << "address of x = " << &x << endl;
9     cout << "z = " << z << '\t' << "address of z = " << &z << endl;
10    cout << "y = " << y << '\t' << "address of y = " << &y << endl;
11
12    z = *y; // Get content from the address stored in y, put it into z
13    cout << endl;
14    cout << "z = " << z << '\t' << "address of z = " << &z << endl;
15    cout << "y = " << y << '\t' << "*y = " << *y << endl;
16    return 0;
17 }
```

Analogy: Name of a Website as a Pointer Variable

anchor text/figure of CSE website (pointer variable)

Departments and Division

SENG comprises 6 departments and 1 division, which are our academic and research pillars spanning crucial engineering disciplines that have pivotal impact on different aspects of our lives and surroundings. They include environmental protection, energy conservation, technological advancement, our health as well as decision-making and analytical skills to navigate the knowledge economy, and even space exploration. It is our mission to contribute by nurturing a wide spectrum of talents in these influential fields.

CBE
Department of Chemical and Biological Engineering

CIVL
Department of Civil and Environmental Engineering

CSE
Department of Computer Science and Engineering

ECE
Department of Electronic and Computer Engineering

IEDA
Department of Industrial Engineering and Decision Analytics

MAE
Department of Mechanical and Aerospace Engineering

About SENG

- HKUST Engineering
- Mission & Vision
- Meet the Dean
- Leadership
- Our People
- Departments & Centers
- Facts & Figures
- Rankings
- Awards
- Jobs@HKUST
- Contact Us

content of pointer variable = CSE web address

<https://cse.ust.hk>

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

```
1 #include <iostream>      /* File: pointer.cpp */
2 using namespace std;
3
4 int main()
5 {
6     int x1 = 10, x2 = 20;
7     int *p1 = &x1;          // p1 now points to x1
8     int *p2 = &x2;          // p2 now points to x2
9
10    *p1 = 5;              // now x1 = 5
11    *p2 += 1000;          // now x2 = 1020
12    *p1 = *p2;            // now *p1 = *p2 = x1 = x2 = 1020, but p1 != p2
13    p1 = p2;              // now p1 and p2 both point to x2
14
15    cout << "x1 = " << x1 << '\t' << "&x1 = " << &x1 << endl;
16    cout << "x2 = " << x2 << '\t' << "&x2 = " << &x2 << endl;
17    cout << "p1 = " << p1 << '\t' << "*p1 = " << *p1 << endl;
18    cout << "p2 = " << p2 << '\t' << "*p2 = " << *p2 << endl;
19
20    return 0;
21 }
```

Example: Pointer and `sizeof()`

```
1 #include <iostream>      /* File: pointer-sizeof.cpp */
2 using namespace std;
3
4 int main()
{
5     char    c = 'A';    char* pc = &c;
6     short   s = 5;     short* ps = &s;
7     int     i = 10;    int* pi = &i;
8     double  d = 5.6;  double* pd = &d;
9
10
11    cout << sizeof(pc) << '\t' << sizeof(*pc) << '\t' << sizeof(&pc)
12        << endl;
13    cout << sizeof(ps) << '\t' << sizeof(*ps) << '\t' << sizeof(&ps)
14        << endl;
15    cout << sizeof(pi) << '\t' << sizeof(*pi) << '\t' << sizeof(&pi)
16        << endl;
17    cout << sizeof(pd) << '\t' << sizeof(*pd) << '\t' << sizeof(&pd)
18        << endl;
19
20    return 0;
21 }
```

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!**

Example: Pointer to Pointer to Pointer ...

```
1 #include <iostream>      /* File: pointer-pointer.cpp */
2 using namespace std;
3
4 int main()
5 {
6     int x = 16;
7     int* xp = &x;          // xp --> x
8     int** xpp = &xp;       // xpp --> xp --> x
9     int*** xppp = &xpp;  // xppp --> xpp --> xp --> x
10
11    cout << "x address      = " << &x      << "    x      = " << x << endl;
12    cout << "xp address     = " << &xp     << "    xp     = " << xp
13                  << "    *xp     = "      << *xp << endl;
14
15    cout << "xpp address    = " << &xpp << "    xpp    = " << xpp
16                  << "    *xpp    = "      << *xpp << "    **xpp   = " << **xpp << endl;
17
18    cout << "xppp address   = " << &xppp << "    xppp   = " << xppp
19                  << "    *xppp   = "      << *xppp << "    **xppp  = " << **xppp
20                  << "    ***xppp = "      << ***xppp << endl;
21    return 0;
22 }
```

Variable, Reference Variable, Pointer Variable

```
1 #include <iostream>      /* File: confusion.cpp */
2 using namespace std;
3
4 int x = 5;                // An int variable
5 int& xref = x;            // A reference variable: xref is an alias of x
6 int* xptr = &x;           // A pointer variable: xptr points to x
7
8 void xprint()
9 {
10    cout << hex << endl; // Print numbers in hexadecimal format
11    cout << "x = " << x    << "\t\tx address = " << &x    << endl;
12    cout << "xref = " << xref << "\t\txref address = " << &xref << endl;
13    cout << "xptr = " << xptr << "\txptr address = " << &xptr << endl;
14    cout << "*xptr = " << *xptr << endl;
15 }
16
17 int main()
18 {
19     x += 1; xprint();
20     xref += 1; xprint();
21     xptr = &xref; xprint(); // Now xptr points to xref
22
23     return 0;
24 }
```

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!**
- **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.

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;`

- 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

```
1 #include <iostream>      /* File: pbv-pointer.cpp */
2 using namespace std;
3
4 void swap(int* x, int* y)
{
    cout << "x = " << x << "\t*x = " << *x << endl;
    cout << "y = " << y << "\t*y = " << *y << endl << endl;
8
9     int temp = *x; *x = *y; *y = temp;
10
11    cout << "x = " << x << "\t*x = " << *x << endl;
12    cout << "y = " << y << "\t*y = " << *y << endl << endl;
13}
14
15 int main()
16 {
17     int a = 10, b = 20;
18     cout << "a = " << a << "\t\t\t&a = " << &a << endl;
19     cout << "b = " << b << "\t\t\t&b = " << &b << endl << endl;
20
21     swap(&a, &b);
22     cout << "a = " << a << "\t\t\ttb = " << b << endl;
23     return 0;
24 }
```

- 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

Example: Euclidean Distance Again — point.h

```
/* File: point.h */  
  
struct Point  
{  
    double x;  
    double y;  
};
```

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

```
1 #include <iostream>      /* File: point-test.cpp */
2 #include "point.h"
3 using namespace std;
4
5 // To compute and print the Euclidean distance between 2 points
6 void print_distance(const Point*, const Point*);
7
8 int main() /* To find the length of the sides of a triangle */
9 {
10     Point a, b, c;
11     cout << "Enter the co-ordinates of point A: "; cin >> a.x >> a.y;
12     cout << "Enter the co-ordinates of point B: "; cin >> b.x >> b.y;
13     cout << "Enter the co-ordinates of point C: "; cin >> c.x >> c.y;
14
15     print_distance(&a, &b);
16     print_distance(&b, &c);
17     print_distance(&c, &a);
18     return 0;
19 }
20 /* g++ -o point-test point-test.cpp point-distance.cpp */
```

Example: Euclidean Distance Again — point-distance.cpp

```
1 #include <iostream>      /* File: point-distance.cpp */
2 #include <cmath>
3 #include "point.h"
4 using namespace std;
5
6 double euclidean_distance(const Point* p1, const Point* p2)
7 {
8     double x_diff = p1->x - p2->x, y_diff = p1->y - p2->y;
9     return sqrt(x_diff*x_diff + y_diff*y_diff);
10 }
11
12 void print_point(const Point* p)
13 {
14     cout << '(' << p->x << ", " << p->y << ')';
15 }
16
17 void print_distance(const Point* p1, const Point* p2)
18 {
19     cout << "Distance between "; print_point(p1);
20     cout << " and "; print_point(p2);
21     cout << " is " << euclidean_distance(p1, p2) << endl;
22 }
```

Example: Student Record Again — student-record.h

```
/* File: student-record.h */
enum Dept { CSE, ECE, MATH };

struct Date
{
    unsigned int year;
    unsigned int month;
    unsigned int day;
};

struct Student_Record
{
    char name[32];
    unsigned int id;
    char gender;
    Dept dept;
    Date entry;
};

// Global constants for department names
const char dept_name[ ][30]
= {"Computer Science", "Electrical Engineering", "Mathematics"};
```

Example: sort-student-record.cpp Again

```
1 #include "student-record.h" /* File: sort-student-record.cpp */
2 #include "student-record-extern.h"
3
4 int main()
{
5
6     Student_Record sr[] = {
7         { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
8         { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
9         { "Cathy", 10000, 'F', ECE, { 2006 , 8 , 20 } } };
10
11    Date d; // Modify the 3rd record
12    set_date(&d, 1980, 12, 25);
13    set_student_record(&sr[2], "Jane", 18000, 'F', CSE, &d);
14
15    sort_3SR_by_id(sr);
16    for (int j = 0; j < sizeof(sr)/sizeof(Student_Record); j++)
17        print_student_record(&sr[j]);
18    return 0;
19 }
20 /* g++ -o sort-sr sort-student-record.cpp student-record-functions.cpp
21    student-record-swap.cpp */
```

Example: student-record-swap.cpp Again

```
1 #include "student-record.h" /* File: student-record-swap.cpp */
2
3 void swap_SR(Student_Record* x, Student_Record* y)
4 {
5     Student_Record temp = *x;
6     *x = *y;
7     *y = temp;
8 }
9
10
11 void sort_3SR_by_id(Student_Record sr[])
12 {
13     if (sr[0].id > sr[1].id) swap_SR(&sr[0], &sr[1]);
14     if (sr[0].id > sr[2].id) swap_SR(&sr[0], &sr[2]);
15     if (sr[1].id > sr[2].id) swap_SR(&sr[1], &sr[2]);
16 }
```

Example: student-record-functions.cpp Again

```
1 #include <iostream> /* File: student-record-functions.cpp */
2 #include "student-record.h"
3 using namespace std;
4
5 void print_date(const Date* date)
6 {
7     cout << date->year << '/'
8         << date->month << '/'
9         << date->day << endl;
10 }
11
12 void print_student_record(const Student_Record* x)
13 {
14     cout << endl;
15     cout.width(12); cout << "name: " << x->name << endl;
16     cout.width(12); cout << "id: " << x->id << endl;
17     cout.width(12); cout << "gender: " << x->gender << endl;
18     cout.width(12); cout << "dept: " << dept_name[x->dept] << endl;
19     cout.width(12); cout << "entry date: "; print_date(&x->entry);
20 }
```

Example: student-record-functions.cpp Again ..

```
21
22 void set_date(Date* x, unsigned int year,
23                 unsigned int month, unsigned int day)
24 {
25     x->year = year;
26     x->month = month;
27     x->day = day;
28 }
29
30 void set_student_record(Student_Record* a, const char name[],
31                         unsigned int id, char gender, Dept dept,
32                         const Date* date)
33 {
34     strcpy(a->name, name);
35     a->id = id;
36     a->gender = gender;
37     a->dept = dept;
38     a->entry = *date;    // struct-struct assignment
39 }
```

Example: student-record-extern.h Again

```
1  /* File: student-record-extern.h */
2
3  void print_date(const Date* );
4  void print_student_record(const Student_Record* );
5
6  void set_date(Date* x, unsigned int, unsigned int, unsigned int);
7  void set_student_record(Student_Record*, const char[],
8                         unsigned int, char, Dept, const Date* );
9
10 void swap_SR(Student_Record*, Student_Record* );
11 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

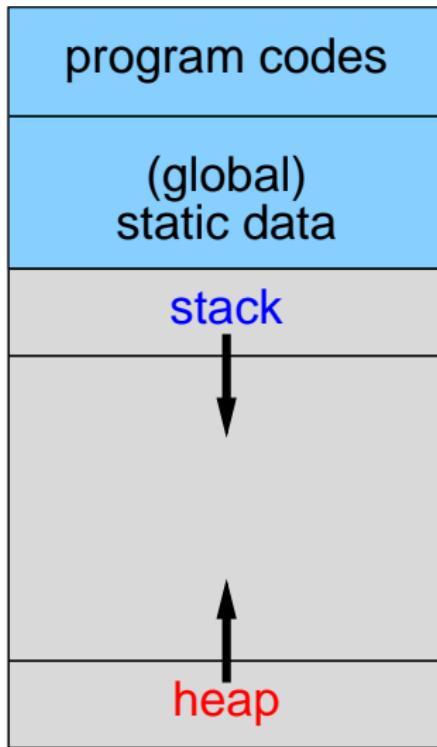
int main()
{
    int x = 5;           // Local int variable
    char s[16] = "hkust"; // Local char array
    return 0;
}
```

- 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



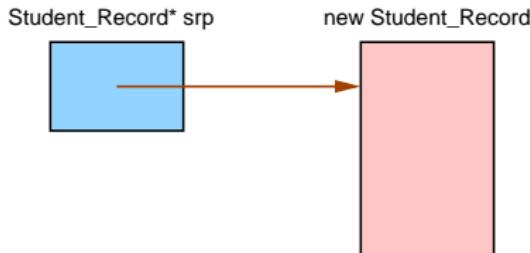
Dynamic Memory Allocation: Operator **new**

Syntax: Dynamic Memory Allocation Using **new**

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

Examples: Use of the **new** Operator

```
int* ip = new int;  
*ip = 5;  
Date 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**.

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 `sdp`, 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.

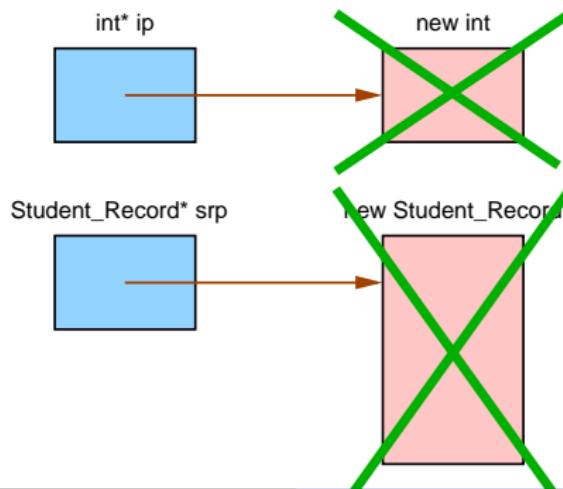
Dynamic Memory Deallocation: Operator **delete**

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.
- Analogy: you keep the old address of a friend who has moved to a new house.
- 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?

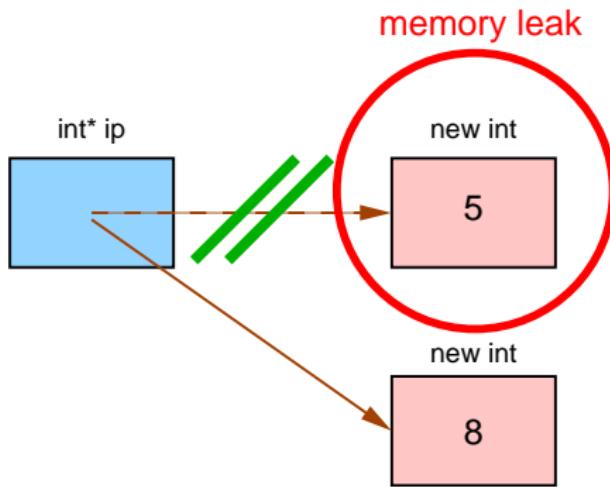
Common Bug II: Memory Leak

- **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**.
- Analogy: your home is full of old stuffs that you don't need anymore. Thus, your room is running out of space. You should properly recycle them.

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 ..

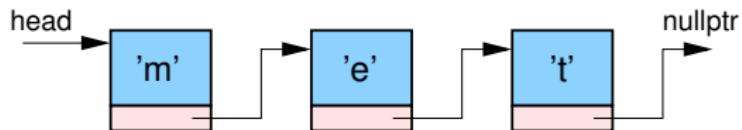
When `swap()` returns,

- 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()`?

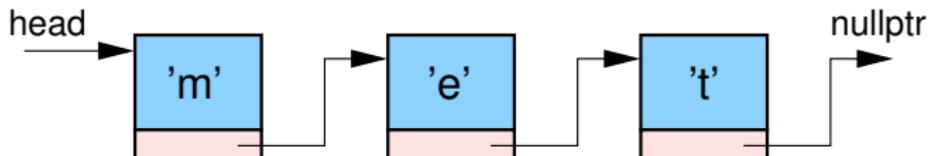
Part V

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_cnode    // C++11 allows default values for members
{
    <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 **nullptr** 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

```
1  /* To create a node */
2  ll_node* p = new ll_node;
3
4  /* To access/modify the data in a node */
5  cout << p->data;
6  cout << (*p).data;
7
8  cin >> p->data;
9  p->next = nullptr;
10
11 /* To set up the head of a linked list */
12 ll_node* head = nullptr; // An empty linked list
13 head = p;               // head points to the node that p points to
14
15 /* To delete a node */
16 delete p;                // Dangling pointer
17 p = nullptr;              // Reset the pointer for safety reason
```

Example: Create the LL-String “met”

```
1 #include "ll_cnode.h" /* File: ll_main.cpp */
2
3 int main() // Create the LL-string "met"
4 { // Create each of the 3 ll_cnodes
5     ll_cnode* mp = new ll_cnode; mp->data = 'm';
6     ll_cnode* ep = new ll_cnode; ep->data = 'e';
7     ll_cnode* tp = new ll_cnode; tp->data = 't';
8
9     // Hook them up in the required order to create the LL
10    mp->next = ep;
11    ep->next = tp;
12    tp->next = nullptr;
13
14    // Traverse the LL and print out the data sequentially
15    for (ll_cnode* p = mp; p; p = p->next)
16        cout << p->data;
17    cout << endl;
18
19    // Clean up
20    delete mp; delete ep; delete tp; return 0;
21 }
```

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 operations, 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 // C++11 allows default values for members
{
    char data;           // Contains useful information
    ll_cnode* next = nullptr; // 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

```
1 #include "ll_cnode.h" /* File: ll_create.cpp */
2 // Create a ll_cnode and initialize its data
3 ll_cnode* ll_create(char c)
4 {
5     ll_cnode* p = new ll_cnode; p->data = c; return p;
6 }
7
8 // Create a linked list of ll_cnodes with the contents of a char array
9 ll_cnode* ll_create(const char s[])
10 {
11     if (s[0] == NULL_CHAR) // Empty linked list due to empty C string
12         return nullptr;
13
14     ll_cnode* head = ll_create(s[0]); // Special case with the head
15
16     ll_cnode* p = head; // p is the working pointer
17     for (int j = 1; s[j] != NULL_CHAR; ++j)
18     {
19         p->next = ll_create(s[j]); // Link current cnode to the new cnode
20         p = p->next; // p now points to the new ll_cnode
21     }
22
23     return head; // The WHOLE linked list can be accessed from the head
24 }
```

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 != nullptr; p = p->next)
        ++length;
    return length;
}
```

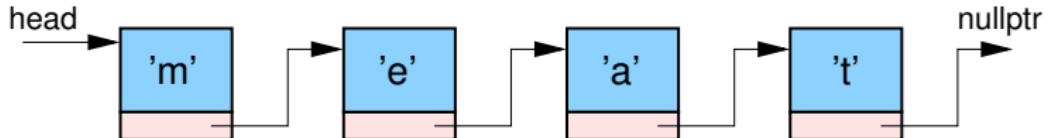
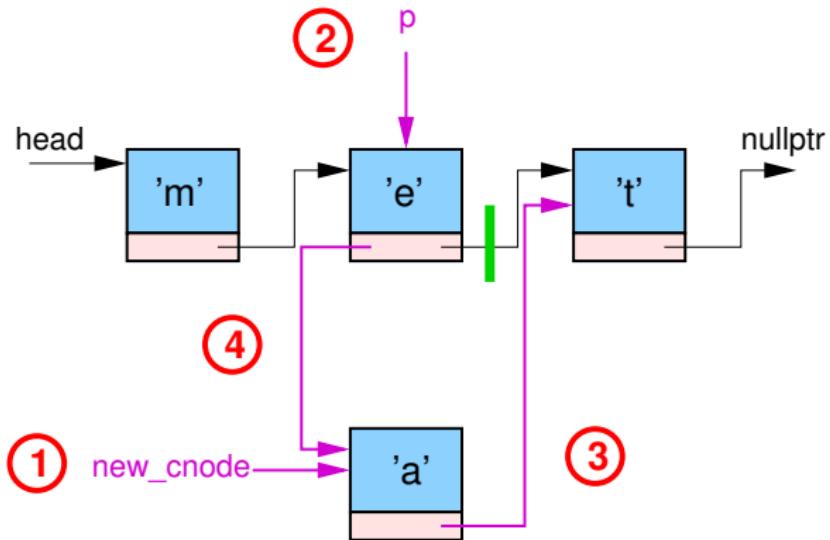
```
#include "ll_cnode.h" /* File: ll_print.cpp */

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

Example: LL-String — ll_search.cpp

```
1 #include "ll_cnode.h" /* File: ll_search.cpp */
2
3 // The returned pointer may be used to change the content
4 // of the found ll_cnode. Therefore, the return type
5 // should not be const ll_cnode*.
6
7 ll_cnode* ll_search(ll_cnode* head, char c)
8 {
9     for (ll_cnode* p = head; p != nullptr; p = p->next)
10    {
11        if (p->data == c)
12            return p;
13    }
14
15    return nullptr;
16 }
```

Example: LL-String — Insertion Algorithm



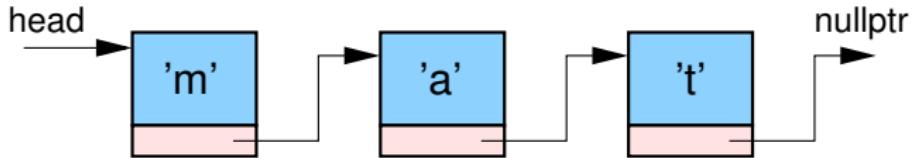
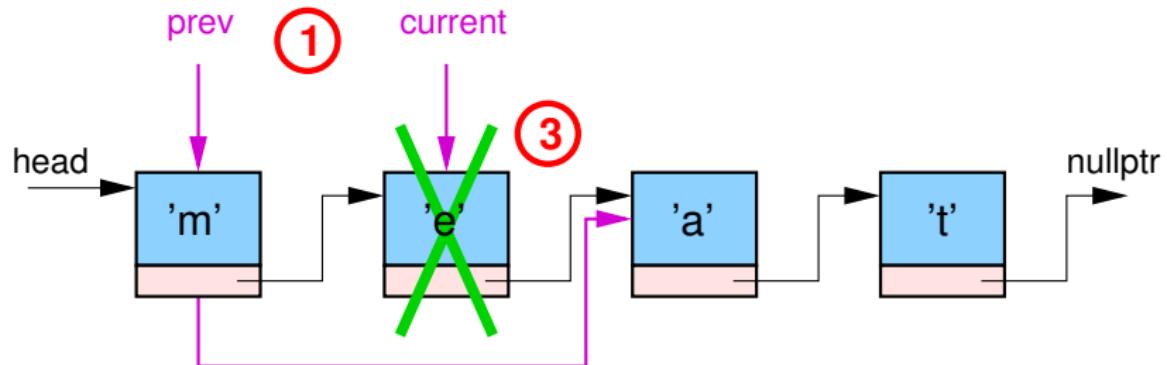
Example: LL-String — ll_insert.cpp

```
1 #include "ll_cnode.h"    /* File: ll_insert.cpp */
2
3 // To insert character c to the linked list so that after insertion,
4 // c is the n-th character (counted from zero) in the list.
5 // If n > current length, append to the end of the list.
6
7 void ll_insert(ll_cnode*& head, char c, unsigned n)
8 {
9     // STEP 1: Create the new ll_cnode
10    ll_cnode* new_cnode = ll_create(c);
11
12    // Special case: insert at the beginning
13    if (n == 0 || head == nullptr)
14    {
15        new_cnode->next = head;
16        head = new_cnode;
17        return;
18    }
19
20}
```

Example: LL-String — ll_insert.cpp ..

```
21     // STEP 2: Find the node after which the new node is to be added
22     ll_cnode* p = head;
23     for (int position = 0;
24         position < n-1 && p->next != nullptr;
25         p = p->next, ++position)
26     ;
27
28     // STEP 3,4: Insert the new node between
29     //             the found node and the next node
30     new_cnode->next = p->next; // STEP 3
31     p->next = new_cnode;      // STEP 4
32 }
```

Example: LL-String — Deletion Algorithm



Example: LL-String — ll_delete.cpp

```
1 #include "ll_cnode.h" /* File: ll_delete.cpp */
2 // To delete the character c from the linked list.
3 // Do nothing if the character cannot be found.
4 void ll_delete(ll_cnode*& head, char c)
5 {
6     ll_cnode* prev = nullptr; // Point to previous ll_cnode
7     ll_cnode* current = head; // Point to current ll_cnode
8
9     // STEP 1: Find the item to be deleted
10    while (current != nullptr && current->data != c)
11    {
12        prev = current;      // Advance both pointers
13        current = current->next;
14    }
15
16    if (current != nullptr) // Data is found
17    { // STEP 2: Bypass the found item
18        if (current == head) // Special case: delete the first item
19            head = head->next;
20        else
21            prev->next = current->next;
22
23        delete current;      // STEP 3: Free up the memory of the deleted item
24    }
25 }
```

Example: LL-String — ll_delete_all.cpp

```
1 #include "ll_cnode.h" /* File: ll_delete_all.cpp */
2
3 // To delete the WHOLE linked list, given its head by recursion.
4 void ll_delete_all(ll_cnode*& head)
5 {
6     if (head == nullptr) // An empty list; nothing to delete
7         return;
8
9     // STEP 1: First delete the remaining nodes
10    ll_delete_all(head->next);
11
12    // For debugging: this shows you what are deleting
13    cout << "deleting " << head->data << endl;
14
15    delete head;           // STEP 2: Then delete the current nodes
16    head = nullptr;        // STEP 3: To play safe, reset head to nullptr
17 }
```

Example: LL-String — ll_test.cpp

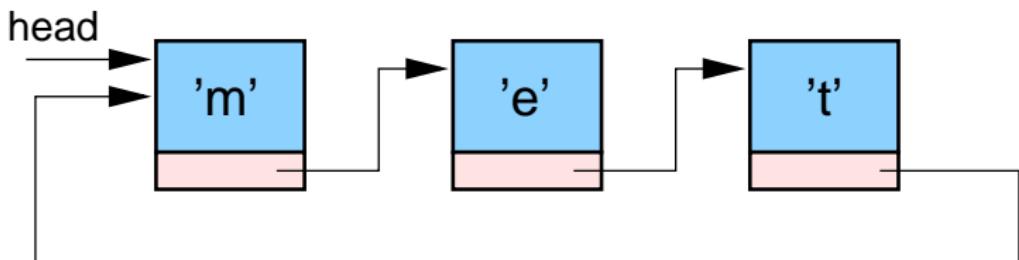
```
1 #include "ll_cnode.h" /* File: ll_test.cpp */
2 int main()
3 {
4     ll_cnode* ll_string = ll_create("met");
5     cout << "length of ll_string = " << ll_length(ll_string) << endl;
6     ll_print(ll_string);
7     ll_print(ll_search(ll_string, 'e'));
8
9     cout << endl << "After inserting 'a'" << endl;
10    ll_insert(ll_string, 'a', 2); ll_print(ll_string);
11    cout << endl << "After deleting 'e'" << endl;
12    ll_delete(ll_string, 'e'); ll_print(ll_string);
13    cout << endl << "After deleting 'm'" << endl;
14    ll_delete(ll_string, 'm'); ll_print(ll_string);
15
16    cout << endl << "After inserting 'e'" << endl;
17    ll_insert(ll_string, 'e', 9); ll_print(ll_string);
18    cout << endl << "After deleting 't'" << endl;
19    ll_delete(ll_string, 't'); ll_print(ll_string);
20    cout << endl << "After deleting 'e'" << endl;
```

Example: LL-String — ll_test.cpp ..

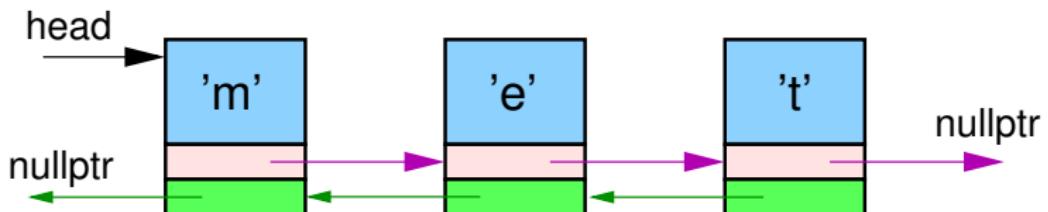
```
21     ll_delete(ll_string, 'e'); ll_print(ll_string);
22     cout << endl << "After deleting 'a'" << endl;
23     ll_delete(ll_string, 'a'); ll_print(ll_string);
24
25     cout << endl << "After deleting 'z'" << endl;
26     ll_delete(ll_string, 'z'); ll_print(ll_string);
27     cout << endl << "After inserting 'h'" << endl;
28     ll_insert(ll_string, 'h', 9); ll_print(ll_string);
29     cout << endl << "After inserting 'o'" << endl;
30     ll_insert(ll_string, 'o', 0); ll_print(ll_string);
31
32     ll_delete_all(ll_string);
33     return 0;
34 }
```

Other Common Variants of Linked List

- Circular Linked List



- Doubly Linked List



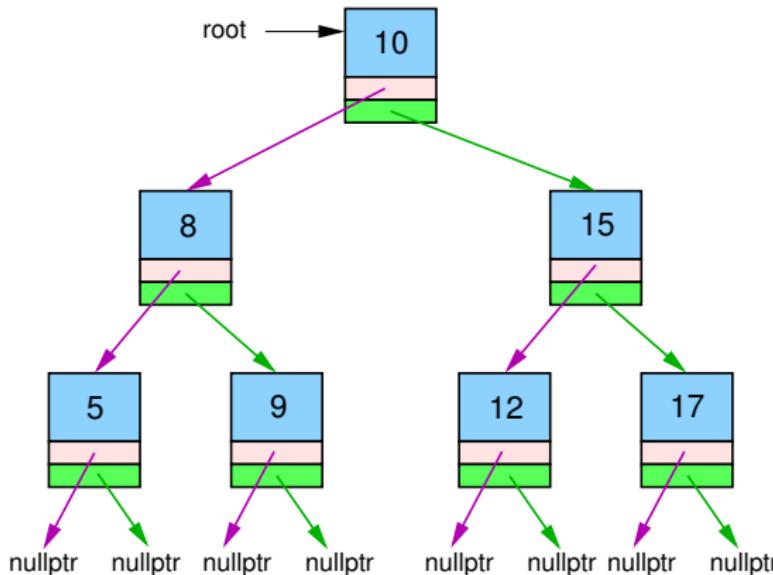
Part VI

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).



Example: Binary Tree — btree.h

```
#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

```
1 #include "btree.h"      /* File: btree-main.cpp */
2
3 int main()
4 {
5     btree_node* root = create_btree_node(10); // Create root node
6
7     root->left = create_btree_node(8);    // Create the left sub-tree
8     root->left->left = create_btree_node(5);
9     root->left->right = create_btree_node(9);
10
11    root->right = create_btree_node(15); // Create the right sub-tree
12    root->right->left = create_btree_node(12);
13    root->right->right = create_btree_node(17);
14
15    print_btree(root); // Print the resulting tree
16
17    delete_btree(root->left);           // Delete the left sub-tree
18    cout << "\n\n\n"; print_btree(root); // Print the resulting tree
19
20 }
```

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

```
1 #include "btree.h"      /* File: btree-create-delete.cpp */
2
3 btree_node* create_btree_node(int data)
4 {
5     btree_node* node = new btree_node;
6     node->data = data;
7     node->left = node->right = nullptr;
8     return node;
9 }
10
11 void delete_btree(btree_node*& root) // By recursion
12 {
13     if (root == nullptr) return;      // Base case
14
15     delete_btree(root->left);      // Recursion on the left subtree
16     delete_btree(root->right);     // Recursion on the right subtree
17     delete root;
18     root = nullptr;
19 }
```

Example: Binary Tree — btree-print.cpp

```
1 #include "btree.h"          /* File: btree-print.cpp */
2
3 void print_btree(const btree_node* root, int depth)
4 {
5     if (root == nullptr)    // Base case
6         return;
7
8     print_btree(root->right, depth+1); // Recursion: right subtree
9
10    for (int j = 0; j < depth; j++)   // Print the node data
11        cout << '\t';
12    cout << root->data << endl;
13
14    print_btree(root->left, depth+1); // Recursion: left subtree
15 }
```

Part VII

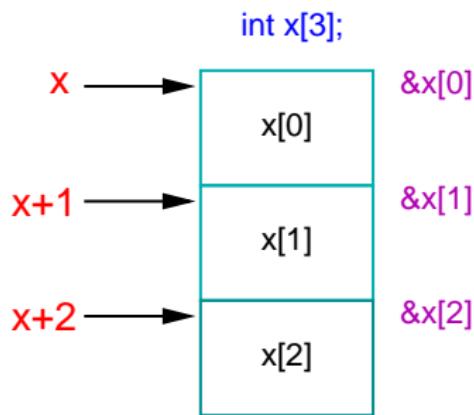
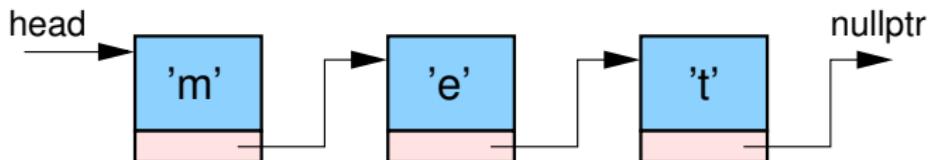
Array as a Pointer

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

Example: Pointer Arithmetic

```
1 #include <iostream>      /* File: pointer-math.cpp */
2 using namespace std;
3
4 int main()
5 {
6     double x = 2.3;      // double is 8-byte
7     double* xp = &x;      // xp points to x
8     cout << &x << endl << xp + 2 << endl << xp - 2 << endl;
9
10    // Nothing disallows you from assigning an integer value
11    // to a pointer variable. Hexadecimal numbers start with 0x.
12    int* yp = reinterpret_cast<int*>(0x14);
13    cout << yp + 1 << endl << yp - 1 << endl;
14
15    // Since addresses around 0x14 may not be accessible to you
16    // Dereferencing them usually leads to runtime error
17    cout << *(yp + 1) << endl << *(yp - 1) << endl;
18    return 0;
19 }
```

The Duality of Array and 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** can be treated as a **const pointer**.
- 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.

```
1 #include <iostream>      /* File: array-by-another-pointer.cpp */
2 using namespace std;
3
4 int main()
5 {
6     int x[] = { 11, 22, 33, 44 };
7     int* y = x; // Both y and x point to the 1st element of array
8
9     // Modify the array through pointer y
10    for (int j = 0; j < sizeof(x)/sizeof(int); ++j)
11        y[j] += 100;
12
13    // Print the array through pointer x
14    for (int j = 0; j < sizeof(x)/sizeof(int); ++j)
15        cout << x[j] << endl;
16    return 0;
17 }
```

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 $\boxed{\text{int } x[] = \{11,22,33\}; \text{int* } xp = x;}$, 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;
}
```

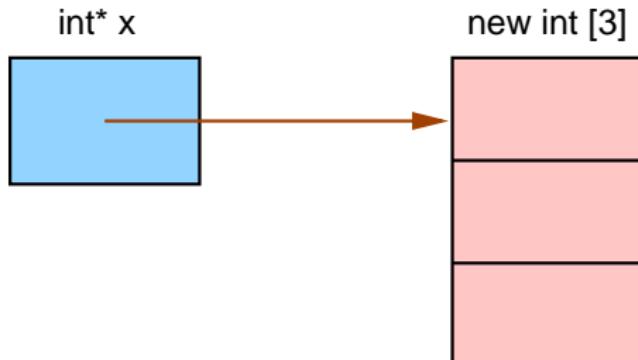
Creation of Dynamic Array: Operator `new` Again

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
```



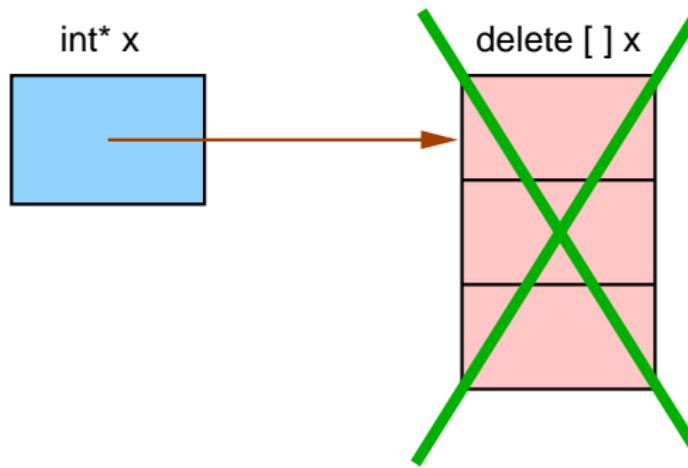
Destruction of Dynamic Array: Operator `delete` Again

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

```
1 #include <iostream>      /* File: dynamic-point-array.cpp */
2 #include "point.h"
3 using namespace std;
4
5 int main()
{
6
7     void print_distance(const Point*, const Point*);
8     int num_points;
9     cout << "Enter the number of points : "; cin >> num_points;
10    Point* point = new Point [num_points]; // Dynamic array of points
11
12    for (int j = 0; j < num_points; ++j) // Input the points
13    {
14        cout << "Enter the x & y coordinates of point #" << j << " : ";
15        cin >> point[j].x >> point[j].y;
16    }
17
18    for (int i = 0; i < num_points; ++i) // Compute distance between 2 points
19        for (int j = i+1; j < num_points; ++j)
20            print_distance(point+i, point+j);
21
22    delete [] point; // Deallocate the dynamic array of points
23    return 0;
24 } /* g++ dynamic-point-array.cpp point-distance.cpp */
```

Example: Dynamic 1D Array ..

```
1 #include <iostream>      /* File: point-distance.cpp */
2 #include <cmath>
3 #include "point.h"
4 using namespace std;
5
6 double euclidean_distance(const Point* p1, const Point* p2)
7 {
8     double x_diff = p1->x - p2->x, y_diff = p1->y - p2->y;
9     return sqrt(x_diff*x_diff + y_diff*y_diff);
10 }
11
12 void print_point(const Point* p)
13 {
14     cout << '(' << p->x << ", " << p->y << ')';
15 }
16
17 void print_distance(const Point* p1, const Point* p2)
18 {
19     cout << "Distance between "; print_point(p1);
20     cout << " and "; print_point(p2);
21     cout << " is " << euclidean_distance(p1, p2) << endl;
22 }
```

Example: C String as Char Pointer

```
1 #include <iostream>      /* File: palindrome.cpp */
2 using namespace std;
3
4 bool palindrome(const char* first, const char* last)
5 {
6     if (first >= last)
7         return true;
8     else if (*first != *last)
9         return false;
10    else
11        return palindrome(first+1, last-1);
12 }
13
14 int main()
15 {
16     const int MAX_LINE_LEN = 255;
17     char s[MAX_LINE_LEN+1];
18     while (cin.getline(s, MAX_LINE_LEN+1, '\n'))
19         cout << booldalpha << palindrome(s, s+strlen(s)-1) << endl;
20     return 0;
21 }
```

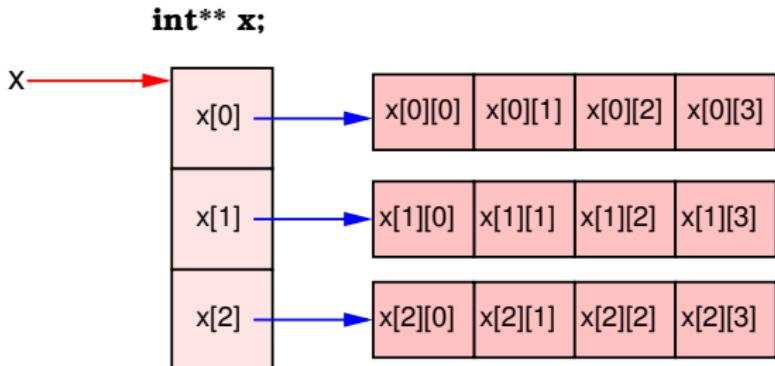
Final Remark on C-string literals

```
1 #include <iostream>      /* File: const-char-pointer.cpp */
2 using namespace std;
3
4 int main()
5 {
6     const char* s1 = "creative"; // Correct way!
7     char* s2 = "smart";        // Error if g++ -pedantic-errors
8
9     /* This only works for char pointer.
10      * E.g., the following are illegal
11      *      const int* x1 = {1, 2, 3};
12      *      int* x2 = {1, 2, 3};
13      */
14
15     cout << s1 << endl;
16     cout << s2 << endl;
17     return 0;
18 }
```

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**:
 - Allocate a **1D array** of **M int*** (int pointers).
 - 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

```
1 #include <iostream>      /* File: 2d-dynamic-array-main.cpp */
2 using namespace std;
3
4 int** create_matrix(int, int);
5 void print_matrix(const int* const*, int, int);
6 void delete_matrix(const int*const *, int, int);
7
8 int main()
9 {
10     int num_rows, num_columns;
11     cout << "Enter #rows followed by #columns: ";
12     cin >> num_rows >> num_columns;
13     int** matrix = create_matrix(num_rows, num_columns);
14
15     // Dynamic array elements can be accessed like static array elements
16     for (int j = 0; j < num_rows; ++j)
17         for (int k = 0; k < num_columns; ++k)
18             matrix[j][k] = 10*(j+1) + (k+1);
19
20     print_matrix(matrix, num_rows, num_columns);
21     delete_matrix(matrix, num_rows, num_columns);
22     matrix = nullptr;      // Avoid dangling pointer
23     return 0;
24 } /* g++ 2d-dynamic-array-main.cpp 2d-dynamic-array-functions.cpp */
```

Example: Operations of a Dynamic 2D Array ..

```
1 #include <iostream>      /* File: 2d-dynamic-array-functions.cpp */
2 using namespace std;
3
4 int** create_matrix(int num_rows, int num_columns) {
5     int** x = new int* [num_rows];          // STEP 1
6     for (int j = 0; j < num_rows; ++j) // STEP 2
7         x[j] = new int [num_columns];
8     return x;
9 }
10
11 void print_matrix(const int* const* x, int num_rows, int num_columns) {
12     for (int j = 0; j < num_rows; ++j)
13     {
14         for (int k = 0; k < num_columns; ++k)
15             cout << x[j][k] << '\t';
16         cout << endl;
17     }
18 }
19
20 void delete_matrix(const int*const * x, int num_rows, int num_columns) {
21     for (int j = 0; j < num_rows; ++j) // Delete is done in reverse order
22         delete [] x[j];           // (compared with its creation)
23     delete [] x;
24 }
```

Example: Relation between Dynamic 2D Array & Pointer

```
1 #include <iostream>      /* File: 2d-dynamic-array-and-pointer.cpp */
2 using namespace std;
3
4 int main()
5 {
6     // Dynamically create an array with 3 rows, 4 columns
7     int** x = new int* [3];      // STEP 1
8     for (int j = 0; j < 3; j++) // STEP 2
9         x[j] = new int [4];
10
11    cout << endl << "Info about x:" << endl;
12    cout << "sizeof(x) :\t" << sizeof(x) << endl << endl;
13    cout << "x\t\t"      << "&x[0]\t\t"      << "&x[0][0]" << endl;
14    cout << x << '\t'      << &x[0]      << '\t' << &x[0][0] << endl << endl;
15    cout << "&x[j]\t\t"        << "x[j]\t\t"
16                  << "&x[j][0]" << '\t'      << "x+j" << endl;
17
18    for (int j = 0; j < 3; j++)
19        cout << &x[j]      << '\t' << x[j] << '\t'
20                      << "&x[j][0]" << '\t' << x+j << endl;
21    return 0;
22 }
```

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.

Part IX

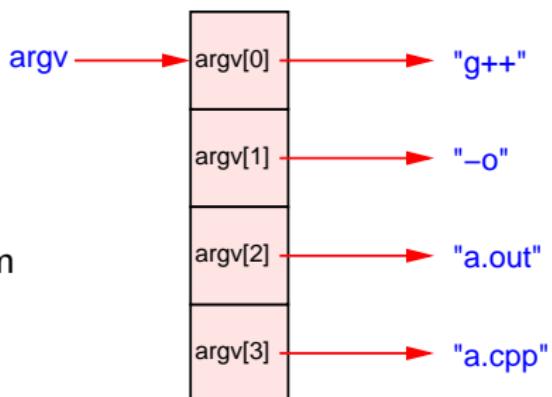
Further Reading I:
Arguments for the main() Function

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

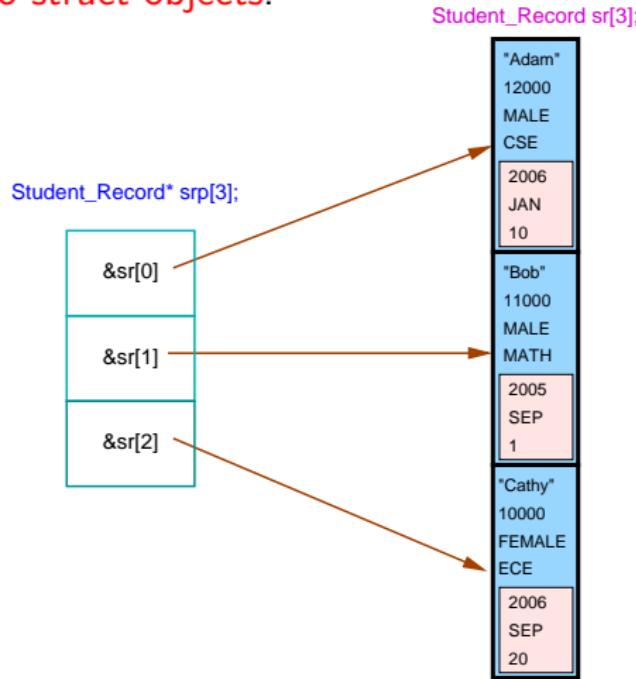
```
1 #include <iostream> /* File: 2d-dynamic-array-main-with-argv.cpp */
2 using namespace std;
3 int** create_matrix(int, int);
4 void print_matrix(const int* const*, int, int);
5 void delete_matrix(int**, int, int);
6
7 int main(int argc, char** argv)
8 {
9     if (argc != 3)
10     { cerr << "Usage: " << argv[0] << " #rows #columns" << endl; return -1; }
11
12     int num_rows = atoi(argv[1]);
13     int num_columns = atoi(argv[2]);
14     int** matrix = create_matrix(num_rows, num_columns);
15
16     // Dynamic array elements can be accessed like static array elements
17     for (int j = 0; j < num_rows; ++j)
18         for (int k = 0; k < num_columns; ++k)
19             matrix[j][k] = 10*(j+1) + (k+1);
20
21     print_matrix(matrix, num_rows, num_columns);
22     delete_matrix(matrix, num_rows, num_columns);
23     matrix = nullptr; // Avoid dangling pointer
24     return 0;
25 } /* g++ 2d-dynamic-array-main-with-argv.cpp 2d-dynamic-array-functions.cpp */
```

Part X

Further Reading II:
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

```
1 #include "student-record.h" /* File: sort-student-record.cpp */
2 #include "student-record-extern.h"
3
4 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 } } };
10
11 Date d; // Modify the 3rd record
12 set_date(&d, 1980, 12, 25);
13 set_student_record(&sr[2], "Jane", 18000, 'F', CSE, &d);
14
15 sort_3SR_by_id(sr);
16 for (int j = 0; j < sizeof(sr)/sizeof(Student_Record); j++)
    print_student_record(&sr[j]);
18 return 0;
19 }
20 /* g++ -o sort-sr sort-student-record.cpp student-record-functions.cpp
21   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

```
1 #include "student-record.h" /* File: sort-student-record-ptr.cpp */
2 void swap_SR_ptr(Student_Record*&, Student_Record*&);
3 void print_student_record(const Student_Record* );
4
5 void sort_3SR_by_id_by_ptr(Student_Record* srp[])
6 {
7     if (srp[0]->id > srp[1]->id) swap_SR_ptr(srp[0], srp[1]);
8     if (srp[0]->id > srp[2]->id) swap_SR_ptr(srp[0], srp[2]);
9     if (srp[1]->id > srp[2]->id) swap_SR_ptr(srp[1], srp[2]);
10 }
11
12 int main()
13 {
14     Student_Record sr[] = {
15         { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
16         { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
17         { "Cathy", 10000, 'F', ECE, { 2009 , 6 , 20 } };
18
19     Student_Record* srp[] = { &sr[0], &sr[1], &sr[2] }; // Array of pointers
20     sort_3SR_by_id_by_ptr(srp);
21
22     for (int j = 0; j < sizeof(srp)/sizeof(Student_Record*); ++j)
23         print_student_record(srp[j]);
24     return 0;
25 } /* g++ sort-student-record-ptr.cpp student-record-ptr-functions.cpp */
```

Example: Sort by Pointers to Struct Objects ..

```
1 #include <iostream> /* File: student-record-ptr-functions.cpp */
2 #include "student-record.h"
3 using namespace std;
4
5 // Swap 2 Student_Record's by their pointers
6 void swap_SR_ptr(Student_Record*& srp1, Student_Record*& srp2)
7 {
8     Student_Record* temp = srp1; srp1 = srp2; srp2 = temp;
9 }
10
11 void print_date(const Date* date)
12 {
13     cout << date->year << '/' << date->month << '/' << date->day << endl;
14 }
15
16 void print_student_record(const Student_Record* x)
17 {
18     cout << endl;
19     cout.width(12); cout << "name: " << x->name << endl;
20     cout.width(12); cout << "id: " << x->id << endl;
21     cout.width(12); cout << "gender: " << x->gender << endl;
22     cout.width(12); cout << "dept: " << dept_name[x->dept] << endl;
23     cout.width(12); cout << "entry date: "; print_date(&x->entry);
24 }
```

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

```
1 #include "student-record.h" /* File: sort-student-record-by-index.cpp */
2 void swap_SR_index(int&, int&);
3 void print_student_record(const Student_Record&);
4
5 void sort_3SR_by_id_by_index(Student_Record sr[], int index[])
6 {
7     if (sr[index[0]].id > sr[index[1]].id) swap_SR_index(index[0], index[1]);
8     if (sr[index[0]].id > sr[index[2]].id) swap_SR_index(index[0], index[2]);
9     if (sr[index[1]].id > sr[index[2]].id) swap_SR_index(index[1], index[2]);
10 }
11
12 int main()
13 {
14     Student_Record sr[] = {
15         { "Adam", 12000, 'M', CSE, { 2006 , 1 , 10 } },
16         { "Bob", 11000, 'M', MATH, { 2005 , 9 , 1 } },
17         { "Cathy", 10000, 'F', ECE, { 2009 , 6 , 20 } };
18
19     int index[ ] = { 0, 1, 2 }; // Array of indices of student records
20     sort_3SR_by_id_by_index(sr, index);
21
22     for (int j = 0; j < sizeof(index)/sizeof(int); ++j)
23         print_student_record(sr[index[j]]);
24     return 0;
25 } // g++ sort-student-record-by-index.cpp student-record-by-index-functions.cpp
```

Example: Sort by Indices to Struct Objects ..

```
1 #include <iostream> /* File: student-record-by-index-functions.cpp */
2 #include "student-record.h"
3 using namespace std;
4
5 // Swap 2 Student_Record's by their indices
6 void swap_SR_index(int& index1, int& index2)
7 {
8     int temp = index1; index1 = index2; index2 = temp;
9 }
10
11 void print_date(const Date& date)
12 {
13     cout << date.year << '/' << date.month << '/' << date.day << endl;
14 }
15
16 void print_student_record(const Student_Record& x)
17 {
18     cout << endl;
19     cout.width(12); cout << "name: " << x.name << endl;
20     cout.width(12); cout << "id: " << x.id << endl;
21     cout.width(12); cout << "gender: " << x.gender << endl;
22     cout.width(12); cout << "dept: " << dept_name[x.dept] << endl;
23     cout.width(12); cout << "entry date: "; print_date(x.entry);
24 }
```