Memory Model

Object-Oriented Programming with C++

Memory Model

What are these variables?

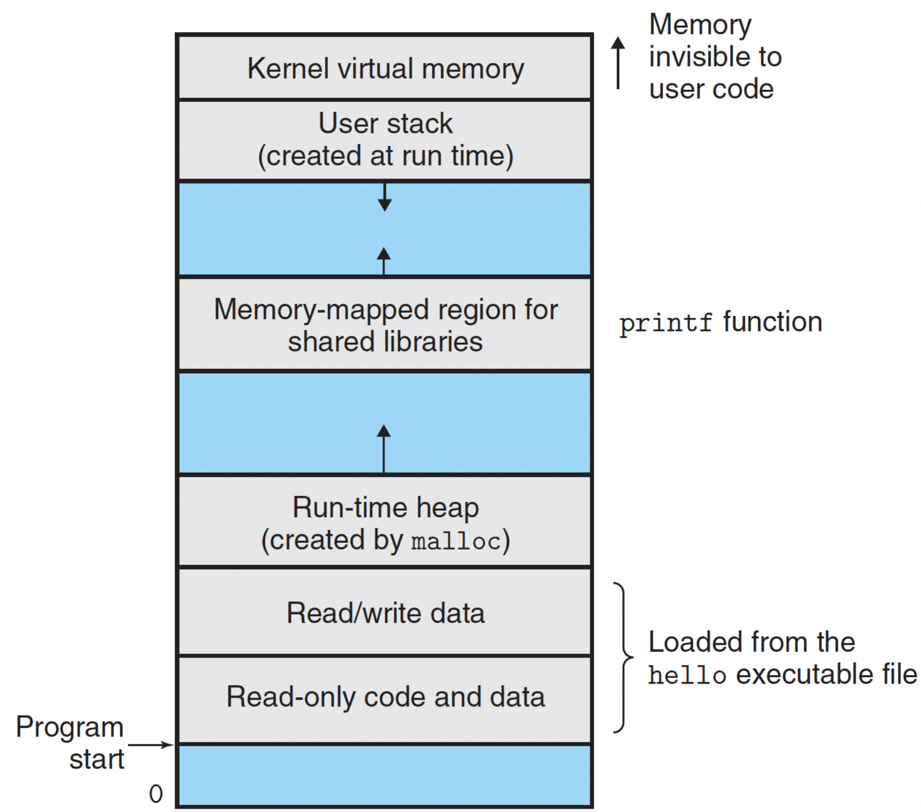
```
int i;                   // global vars.
static int j;            // static global vars.

void f()
{
  int k;                 // local vars.
  static int l;          // static local vars.

  int *p = malloc(sizeof(int)); // allocated vars.
}
```

Where are they in memory?

- stack
  - local vars
- heap
  - dynamically allocated vars.
- code/data
  - global vars
  - static global vars
  - static local vars



Global vars

- vars defined outside any functions
- can be shared btw .cpp files
- extern

Extern

- extern is a declaration says there will be such a variable somewhere in the whole program
- “such a” means the type and the name of the variable
- global variable is a definition, the place for that variable

Static

- static global variable inhibits access from outside the .cpp file
- so as the static function

Static local vars

- static local variable keeps value in between visits to the same function
- is initialized at its first access

Static

- for global stuff:
  - access restriction
- for local stuff:
  - persistence

Pointers to Objects

Pointers to objects

```
string s = "hello";
string* ps = &s;
```

Operators with pointers

- get address

```
ps = &s;
```

- get the object

```
(*ps).length()
```

- call the function

```
ps->length()
```

Two ways to access

- `string s;`
  - s is the object itself
  - At this line, object s is created and initialized
- `string *ps;`
  - ps is a pointer to an object
  - the object ps points to is not known yet.

Assignment

```
string s1, s2;
s1 = s2;

string *ps1, *ps2;
ps1 = ps2;
```

Reference

Defining references

- References are a new data type in C++

```
char c;         // a character
char* p = &c;   // a pointer to a character
char& r = c;    // a reference to a character
```

Defining references

- `type& refname = name;`
  - For ordinary variable definitions
  - An initial value is required
- `type& refname`
  - In parameter lists or member variables
  - Binding defined by caller or constructor

References

- Declares a new name for an existing object

```
int X = 47;
// Y is a reference to X, X and Y now refer to
// the same variable
int &Y = X;

cout << "Y = " << Y;  // prints Y = 47
Y = 18;
cout << "X = " << X;  // prints X = 18
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
  - In definition

```
int x = 3;
int& y = x;
const int& z = x;
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
  - As a function argument

```
void f (int& x);
f(y); // initialized when function is called
```

Rules of references

- Bindings don’t change at run time, unlike pointers
- Assignment changes the object referred-to

```
int& y = x;

y = z; // Change value of x to value of z.
```

Rules of references

- The target of a non-const reference must be an *lvalue*.

```
void func (int &);

func (i * 3); // Warning or Error!
```

Type restrictions

- No references to references
- No pointers to references, but reference to pointer is ok

```
int&* p; // illegal
void f(int*& p); // ok
```

- No arrays of references

Pointers vs. References

Pointers

- independent of the bound object, can be uninitialized
- can be bound to a different object
- can be set to null

References

- dependent on the bound object, just an *alias*, must be initialized
- can't be rebound.
- can't be null

Dynamically Allocated Memory

Dynamic memory allocation

- *new* expression

```
new int;
new Stash;
new int[10];
```

- *delete* expression

```
delete p;
delete[] p;
```

`new` and `delete`

- Similar to *malloc*, `new` is the way to allocate memory as a program runs. Pointers become the only access to that memory.
- Similar to *free*, `delete` enables you to return memory to the memory pool when you are finished with it.
- Besides that, `new` and `delete` ensure the right calling of Ctor/Dtor for objects.

Dynamic arrays

- The new operator returns the address of the first element of the block.

```
int *psome = new int[10];
```

- The presence of the brackets tells the program that it should free the whole array, not just the element

```
delete[] psome;
```

The *new-delete* mechanism

```
int *p = new int;
int *a = new int[10];

Student *q = new Student();
Student *r = new Student[10];

delete p;
delete[] a;

delete q;
delete r;
delete[] r;
```

Tips for *new* and *delete*

- Don't mix-use `new/delete` and `malloc/free`.
- Don't `delete` the same block of memory twice.
- Use `delete` (no brackets) if you've used `new` to allocate a single entity.
- Use `delete[]` if you've used `new[]`.
- `delete` the *null pointer* is safe (nothing happens).

Constant

const

- declares a variable to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++;    // illegal!

int y = x;          // ok, copy const to non-const
y = x;              // ok, same thing
const int z = y;    // ok, const is safer
```

Constants

- Constants are like variables
  - Observe scoping rules
  - Declared with `const` type modifier
- A const in C++ defaults to *internal linkage*
  - the compiler tries to avoid creating storage for a const, holding the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

- Compile time constants are entries in compiler symbol table, not really variables.

```
const int bufsize = 1024;
```

- Value must be initialized
- Unless you make an explicit extern declaration:

Run-time constants

- const value can be exploited

```
const int class_size = 12;
int finalGrade[class_size]; // ok

int x;
cin >> x;

const int size = x;
double classAverage[size]; // error
```

Pointers with `const`

`p: 0xaffefado` ==> `a: [53,54,55]`

```
int a[] = {53,54,55};

int * const p = a;  // p is const
*p = 20;        // OK
p++;            // ERROR

const int *p = a;   // (*p) is const
*p = 20;        // ERROR!
p++;            // OK
```

What are these?

```
string s( "Fred" );
const string* p = &s;
string const* p = &s;
string *const p = &s;
```

Pointers and constants

```
int i;
const int ci = 3;

int* ip;
const int* cip;

ip = &i;
ip = &ci; // Error
cip = &i;
cip = &ci;

*ip = 54;  // always legal
*cip = 54; // never legal
```

String literals

```
char* s = "Hello, world!";
char a[] = "Hello, world!";
```

- `s` is a pointer initialized to point to a string constant
- This is actually a `const char* s` but compiler accepts it without the const
- Don't try to change the character values (undefined behavior)

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);
int a = 15;
f(&a); // ok
const int b = a;

f(&b); // ok
b = a + 1; // Error!
```

- You cannot treat a constant object as non-constant without an explicit cast `const_cast`

Passing by const value?

```
void f1 (const int i) {
  i++; // illegal: compile-time error
}
```

Returning by const value?

```
int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // works fine
  int k = f4(); // this works fine too
}
```

Passing addresses

- Passing large objects are expensive.
- Better to pass by address, using a pointer or a reference.
- Make it `const` whenever possible to prevent unexpected modification.